Leveraging Adaptive I/O to Optimize Collective Data Shuffling Patterns for Big Data Analytics

Nicolae, Bogdan; Costa, Carlos H. A.; Misale, Claudia; Katrinis, Kostas; Park, Yoonho. Leveraging Adaptive I/O to Optimize Collective Data Shuffling Patterns for Big Data Analytics. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. PP (99) pp: 1-13. DOI: 10.1109/TPDS.2016.2627558

Leveraging Adaptive I/O to Optimize Collective Data Shuffling Patterns for Big Data Analytics

Bogdan Nicolae*, Carlos H. A. Costa†, Claudia Misale‡, Kostas Katrinis*, Yoonho Park†
*IBM Research, Ireland. Email: {bogdan.nicolae,katrinisk}@ie.ibm.com
†IBM T.J. Watson Research Center, USA. Email: {chcost,yoonho}@us.ibm.com
‡University of Torino, Italy. Email: misale@di.unito.it

Abstract—Big data analytics is an indispensable tool in transforming science, engineering, medicine, health-care, 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. In this context, data shuffling, a particularly difficult transformation pattern, introduces important challenges. Specifically, data shuffling is a key component of complex computations that has a major impact on the overall performance and scalability. Thus, speeding up data shuffling is a critical goal. To this end, state-of-the-art solutions often rely on overlapping the data transfers with the shuffling phase. However, they employ simple mechanisms to decide how much data and where to fetch it from, which leads to sub-optimal performance and excessive auxiliary memory utilization for the purpose of prefetching. The latter aspect is a growing concern, given evidence that memory per computation unit is continuously decreasing while interconnect bandwidth is increasing. This paper contributes a novel shuffle data transfer strategy that addresses the two aforementioned dimensions by dynamically adapting the prefetching to the computation. We implemented this novel strategy in Spark, a popular in-memory data analytics framework. To demonstrate the benefits of our proposal, we run extensive experiments on an HPC cluster with large core count per node. Compared with the default Spark shuffle strategy, our proposal shows: up to 40% better performance with 50% less memory utilization for buffering and excellent weak scalability.

Index Terms—distributed systems, big data analytics, Spark, data shuffling, scalable I/O, memory efficient concurrent data transfers, I/O load balancing, elastic buffering

1 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 *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.

With large-scale data centers employed by both cloud providers and supercomputing facilities getting closer to Exascale, the increasing core count per node [4] leads to a high degree of intra-node parallelism, which in turn makes it difficult to share in-memory data. In addition, memory becomes a precious resource: prices have stopped dropping for the past years, less and less memory per core is available, 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. Furthermore, it is also important to note that many users increasingly rely on cloud computing to provision computational resources. In this context, memory is shared with other applications as part of multi-tenancy and its utilization is a decisive factor in the operational costs incurred by the pay-as-you-go model, which further increase its preciousness. Thus, running big data analytics efficiently at scale implies an important trade-off: support user-friendly programming models that deliver high performance and scalability while minimizing 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, sort, groupby). With respect to performance and scalability, data shuffling is challenging because it involves complex all-to-all communication patterns, either between local aggregation processes that share the same node or between remote tasks on different nodes that need to communicate over the network. This 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 needed for user data.

Thus, adapting the shuffle strategy to address the issues mentioned above is paramount. This paper contributes with a novel data transfer strategy that is specifically designed to scale both horizontally and vertically to a large number of aggregation tasks under tight memory constraints. It extends our previous preliminary work [5] that formulates the problem of data shuffling under tight memory constraints and outlines a series of design principles to address it: load balancing of fetch requests using executor-level coordination, prioritization based on locality and responsiveness, static circular allocation of initial requests, elastic adjustment of in-flight limit, shuffle block aggregation and dispersal using in-flight increment.

1.1 Contributions

Starting from this preliminary work, we added the following new contributions.

First, we provide in Section 4 more context and detail for the design principles outlined in [5]. Specifically, in our previous work we focused only on remote data transfers, assuming that co-located processing elements on the same node can trivially access each other's shuffle blocks. In this work, we consider a complete solution that is able to differentiate between local and remote data sources. Since co-location of processing elements is a standard practice for multi-core nodes, this extension is critical in a real-world setup. Furthermore, based on the revised design principles, we introduce a series of algorithms that illustrate concretely a possible implementation.

Second, we show in Section 5 how to materialize these design principles and algorithms in practice using a reference prototype implementation that is integrated in the Spark framework.

Third, we run extensive experiments at various scales using state-of-the-art HPC infrastructure with large core count per node, multiple memory configurations and two benchmarking applications. These experiments are discussed in Section 6 and show significant performance gain and lower memory utilization compared with the default Spark shuffle strategy.

2 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 [6], [7].

In big data analytics, data shuffling is a key component of large-scale data aggregations. One widely-known 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*, *sortByKey*, *coalesce*, *cogroup*, *join*, etc. 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*.

In a multi-core setup, many reducers will be co-located on the same node. As a consequence, reducers are often grouped together to share the same memory space through a master process, which we refer to as *executor*. In this case, the shuffle blocks offered by the same executor can be trivially accessed by all its reducers. Ideally, only one executor should be deployed per node to maximize this potential. However, for practical reasons (e.g. NUMA awareness, limitations of underlying technologies such as the amount of threads and memory that can be managed by a single Java Virtual Machine), it is not always feasible to use a single executor per node. Thus, even if two reducers share the same node, they may still need to use communication channels to fetch the shuffle blocks. Such communication channels introduce extra overhead, but this overhead is still lower than using a communication channel over the network between different nodes. Therefore, we differentiate between the case when a reducer needs to fetch shuffle blocks from a co-located executor different than its own (*local source*) and from an executor hosted by a remote node (*remote source*).

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 reducers. Since data transfers are performed concurrently, this creates a complex all-to-all parallel communication pattern that puts a significant burden both at local level (fetching from local sources) and on the networking infrastructure (fetching from remote sources). Thus, lack of coordination between the reducers can quickly lead to I/O bottlenecks, which in turn 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 as a buffer to accumulate shuffle blocks. Spark implements this limited buffer as an upper bound on the amount of shuffle data that can be in transit from other reducers at any point in time. This limit is independently applied to each reducer. In the worst case (i.e. when no shuffle blocks are consumed), only up to the maximum amount of permitted in-flight shuffle data can accumulate. Thus, 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 let each reducer accumulate as many shuffle blocks as possible in the background, 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.

3 Related work

How to optimize the performance of big data applications has been extensively studied in the context of MapReduce. Vertical scalability issues are explored in [8]. With respect to I/O, Ren et al. [9] conclude that improving data locality has little potential to improve I/O performance. They suggest in-memory storage, potentially in form of a DSM (distributed shared memory) as an alternative to disk storage, which is the direction adopted by Spark. Further data locality issues are explored in [10]. Overlapping the map phase with the reduce phase efficiently such that reducers do not lock out resources when idle is explored in [11]. Since data shuffling is acknowledged to be expensive in terms of I/O and network traffic, some approaches propose the analysis of the shuffle phase in order to extract interesting properties that enable shuffle avoidance [12]. Some studies show that CPU can also become a major bottleneck in big data analytics [13].

A particularly sensitive issue that directly impacts the performance of MapReduce jobs is the configuration: there are more than 70 parameters whose optimal value is difficult to establish. To this end, approaches such as MRONLINE [14], [15] aim to dynamically converge to an optimal configuration by constantly adapting the parameters during runtime.

With respect to the storage layer and user data, there are several studies that focus particularly on HDFS [16], [17]: metadata, file access patterns create, read, write, delete, etc. Improved concurrency control through multi-versioning can improve I/O data throughput significantly under concurrency compared with HDFS, as demonstrated by BlobSeer [18]. In-memory caching as an additional layer on top of the storage layer is also gaining increasing attention recently [19]. RDMA-related optimizations are proposed in SOR-HDFS [20], which aims to improve the HDFS write operations. Such approaches can also be complemented by several cloud-specific optimizations related to storage elasticity [21], [22].

With respect to data shuffling itself, the problem has been explored from multiple perspectives. Theoretical consideration was given in [23], where the authors present upper and lower bounds on the parallel I/O complexity of the shuffle phase, bounding the worst-case performance loss of the MapReduce approach in terms of I/O-efficiency. Shared environment optimizations for Hadoop MapReduce based on pre-fetching and pre-shuffling were explored in [24]. In-memory optimizations on NUMA multi-core systems was explored in [25], where a series of techniques such as thread-binding, NUMA-aware thread allocation and relaxed global coordination were demonstrated effective when compared to naive shuffle strategies. 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 [26] and Spark [27]. How to improve shuffle block transfer time and allow scheduling policies at the transfer level (e.g. prioritizing a transfer over other transfers) was explored in [28]. Furthermore, optimizations that are orthogonal to data transfers can be an effective complement: compression [29], [30], natural data redundancy [31], shuffle file consolidation [29].

This paper focuses on high performance, scalability and memory efficiency for data shuffling in the context of big data analytics over high end infrastructure. To our best knowledge, we are the first to focus on these aspects simultaneously.

4 An Adaptive Memory-Efficient Shuffle Block Transfer Proposal

This section details our proposal for an adaptive shuffle block proposal. We focus on two aspects: (1) how a reducer selects the source where to get new shuffle blocks from; (2) when and how many shuffle blocks to fetch from the source. Note that each executor has two different roles: as a coordinator for its reducers and as a source that serves fetch requests. For brevity, we will refer to an executor in the latter role simply as *source*.

4.1 Shuffle block source selection

We assume that reducers prioritize fetching shuffle blocks from local sources over remote sources, which is not only done for performance reasons but also scalability: it alleviates the pressure on the networking infrastructure and helps

conserve bandwidth under concurrency. Thus, as long as there are still local sources left that can offer shuffle blocks, reducers prefer to fetch from them. However, most of the time, there will be more than one local or remote source available to choose from, so there is a need to differentiate between them. To this end, we introduce three criteria for selection, detailed below.

4.1.1 Load balancing of data transfers using executor-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 the 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 executor-level using shared in-memory data structures that keep track of the total amount of in-flight data generated by all reducers under the same executor. In other words, each executor creates a local view of the load of all sources (how much did my reducers ask from whom) which is opposite to a source-based perspective (who asked how much from me). This local view is then used by a reducer to prioritize fetching from the source that is the least loaded, which improves load balancing. While this may not be globally optimal, it is an effective compromise given the large number of co-located reducers per executor.

4.1.2 Prioritization based on source responsiveness

Load balancing alone is not enough to mitigate I/O bottlenecks: even if a source does not need to serve a lot of requests, it can still be less responsive than a heavier loaded one. This can happen because of multiple reasons: starvation due to unfair allocation of I/O bandwidth, high CPU utilization during the computation, heterogeneity of the computation, etc. Furthermore, since load is measured locally from the perspective of the reducers sharing the same executor, it can happen that a source looks to them the least loaded, but in reality it is heavily loaded at global level (e.g., when all reducers simultaneously access the same source because they all issued pending requests to other sources). Thus, we propose to measure for all sources how much time the reducers block waiting for their shuffle blocks. Again, this is a local view from the receiver perspective. Based on this view, a moving average can be calculated for each source based on the most recent shuffle blocks. This effectively enables the selection strategy to *adapt to the particular situation of each source independently* (e.g., a source may look unresponsive to a fast reducer but could be considered responsive otherwise).

4.1.3 Static circular load-balancing of the initial data transfers

One important standing issue is how to assign the initial fetch requests to the sources: since there is no historical information about waiting times and no reducer 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 or if two sources exhibit similar load and responsiveness, then each reducer prefers the source executor that is the closest successor to its own executor. Any predefined circular ordering of the executors can be used, as long as all reducers from all nodes agree on it. Using this approach, reducers belonging to different executors will prefer fetching from different sources with high probability, which reduces the risk for I/O bottlenecks without using any form of synchronization.

4.1.4 Combining the selection criteria into a single metric

To materialize the selection criteria presented above, we introduce a scoring mechanism that combines the three dimensions into a single metric. Specifically, we obtain a derived metric for each source by multiplying the amount of in-flight data from it with the responsiveness measured as average wait. A reducer always prefers the source with the minimum derived metric, but may need to differentiate using the static circular order criteria when there is no unique minimum. This is illustrated in Algorithm 1: each reducer belonging to executor id invokes the RECOMMEND_SOURCE primitive to obtain an optimal source executor min_i from which to fetch new shuffle blocks when needed. We use the following notations: BlockL is the list of remaining shuffle blocks to be fetched by the reducer. AvgKWait is the moving average on the wait time (measured in ms) caused by the last K shuffle blocks fetched from executor i by all reducers of executor id. InFlight is the amount of bytes already in transit from the executor i. Order holds the position of the executor in the static circular pre-defined order. One is added to AvgKWait to make sure InFlight is not ignored when the average wait time for shuffle blocks is zero.

Algorithm 1 Selection algorithm for N sources

```
1:  function RECOMMEND_SOURCE(BlockL)
2:      min_i ← Head[N]
3:      for all i ∈ N such that i holds any b ∈ BlockL do
4:          Score[i] ← (AvgKWait[i] + 1) · InFlight[i]
5:          if (i and id local but min_i remote) then
6:              min_i ← i
7:          else if (Score[i] < Score[id]) or (Score[i] =
    Score[id] and Order[i] > Order[id] and Order[min_i] <
    Order[id]) then
8:              min_i ← i
9:          end if
10:     end for
11:     return min_i
12: end function
```

4.2 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 source 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 source is used: a trivial data transfer planning strategy would simply call RECOMMEND_SOURCE and 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.

4.2.1 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 the executors in order to reduce the risk of fluctuations and unresponsiveness that may happen during the data transfers. 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, RECOMMEND_SOURCE is repeatedly invoked to schedule to fetch requests. Since the in-flight size of the selected source changes after each invocation, a different source is returned each time with high probability. This enables multiple parallel requests to different sources, which reduces the risk of I/O bottlenecks.

4.2.2 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 shorter than the average, then its soft 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. We summarize this process in Algorithm 2.

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.

Algorithm 2 Elastic adjustment of reducer in-flight limit

```
1: function CONSUME_SHUFFLE_BLOCK
2:     B ← dequeue next shuffle block
3:     inFlight ← inFlight − Size[B]
4:     if fetchWait > avgFetchWait and softInFlight < hardInFlight then
5:         softInFlight += incInFlight
6:     else if softInFlight > 2 · incInFlight then
7:         softInFlight -= incInFlight
8:     end if
9:     update avgFetchWait
10:    while inFlight < softInFlight do
11:        i ← RECOMMEND_SOURCE(BlockL)
12:        async fetch up to (softInFlight - inFlight) from i
13:        update inFlight, BlockL
14:    end while
15:    return B
16: end function
```

5 Implementation

Given its enormous traction in production analytics platforms and solutions, we decided to use Spark as the reference runtime to evaluate our techniques. In this section, we provide a brief overview of Spark and its default shuffle strategy, after which we describe a prototype implementation of our approach that we integrated with Spark 1.6.0.

Spark is a big data analytics framework that facilitates the development of multi-step data pipelines using a directed acyclic graph (DAG) as a runtime construct. The user implements a driver program that describes the high-level control flow of the application, which relies on two main parallel programming abstractions: (1) *resilient distributed datasets* (RDDs), a partitioned data structure hosting the data itself; and (2) parallel operations on the RDDs.

An RDD is a read-only, resilient collection of objects partitioned across multiple nodes that holds provenance information (referred to as *lineage*) and can be rebuilt in case of failures by partial recomputation from ancestor RDDs. An RDD can be created in three ways: (1) by using the implicit partitioning of an input file stored in the underlying distributed file system (e.g., HDFS); (2) by explicit partitioning of a native collection (e.g. array); or (3) by applying a transformation to an already existing RDD. Furthermore, each RDD is by default *lazy* and *ephemeral*. The lazy property has the same meaning as in functional programming and refers to the fact that an RDD is computed only when needed. Ephemeral refers to the property that once an RDD actually gets materialized, it will be eventually discarded from memory. However, since RDDs might be repeatedly needed during computations, the user can explicitly mark them as persistent, which moves them in a dedicated cache for persistent objects.

The architecture of the Spark runtime, depicted in Figure 1, is specifically designed to support the above model. A master node coordinates a set of worker nodes, each of

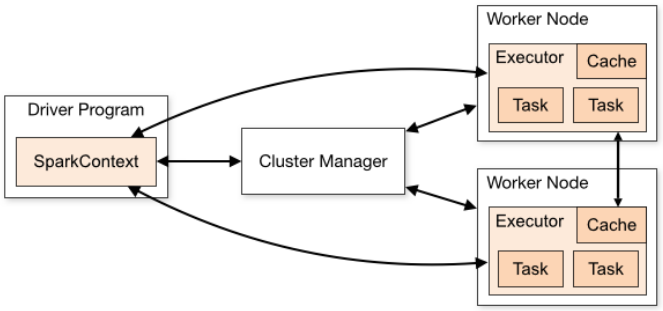


Fig. 1. High-level architecture of the Spark runtime

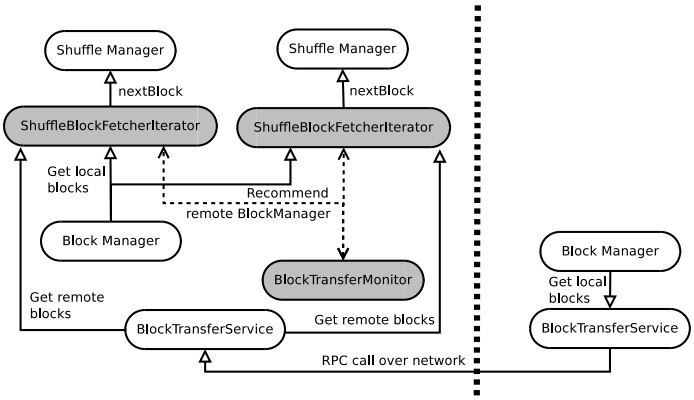


Fig. 2. Adaptive shuffle block transfer as integrated in the Spark shuffle logic. New and modified components are highlighted with a dark background color.

which run executors that parallelize the tasks originating from the execution of RDD operations (termed transformations or actions in Spark nomenclature). For the purpose of this work, we assume that each worker launches a single executor. Thus we will use the term "worker" and "executor" to mean the same thing. All ephemeral RDDs live for the duration of the computation distributed among the memory allocated to the workers. Furthermore, each worker allocates a separate cache memory pool that is reserved for persistent RDDs.

Data shuffling in Spark is implemented as a pull mechanism, which has several important consequences: (1) a reduce task cannot start before all its intermediate data is available in the collective memory of the executors that generated it; (2) the intermediate data must stay available until requested with no information of when this might happen; (3) each reduce task is responsible to decide when and what shuffle blocks to pull.

Figure 2 illustrates the data flow diagram for the shuffle blocks and the interactions between the various components that make up the shuffle logic.

Specifically, the data flow works as follows: the shuffle blocks are accumulated in the background through the *ShuffleBlockFetcherIterator*, which is used by the *Shuffle Manager* to consume the shuffle blocks. First, the *ShuffleBlockFetcherIterator* fetches the local blocks from the *Block Manager*, which is the entity that plays the role of the source. While the local blocks are being consumed, it initiates asynchronous transfers for the rest of blocks that belong to the other executors. This is done through the *BlockTransferService*, which handles all communication, whether local or over the network, with its remote counterpart. In the original implementation, there is no *BlockTransferMonitor*. This is an additional component added by our approach, which also re-implements the *ShuffleBlockFetcherIterator*. To highlight this better, we use a darker background for these components in Figure 2.

Default static shuffle strategy: The default *ShuffleBlockFetcherIterator* uses a static strategy to pull the remote shuffle blocks. Specifically, for each reduce task it precomputes a data transfer plan according to the following scheme: all fetch requests addressed to the same executor are packed together in a set of requests that are not larger than one fifth of the in-flight limit. The rationale of using one fifth is purely empiric and is based on the intuition that spreading the fetch requests over multiple executors reduces the risk of bottlenecks compared to the case when all requests are issued to the same executor. Then, all requests addressed to all executors are randomly shuffled and placed in a queue, in order to achieve a rough form of load balancing. Initially, a number of requests that add up to the in-flight limit are dequeued and the amount of in-flight data is monitored. Then, every time a shuffle block was received, once enough room for the head request in the queue exists (i.e., in-flight data smaller than one fifth of the in-flight limit), the request is dequeued and processed. The process repeats until all requests were issued. Independently, the reduce tasks wait on a separate queue where the shuffle blocks accumulate in order to consume the intermediate data.

Adaptive shuffle strategy: To adopt the principles proposed in Section 4, we implemented a new dynamic scheme that builds requests on the fly and eliminates the need for a request queue: once enough room (i.e., in-flight increment) is available to construct a new request, the *ShuffleBlockFetcherIterator* asks the *BlockTransferMonitor* to recommend a source executor. The *BlockTransferMonitor* is a new component responsible to implement the RECOMMEND_SOURCE primitive. To this end, it keeps track of the relevant metrics (in-flight data amount and responsiveness for each other source) and builds the local view that is used to calculate the combined score. Using the recommendation, the *ShuffleBlockFetcherIterator* constructs a new request to that executor by packing together as many shuffle blocks as possible up to the in-flight increment. This process is repeated until the elastic in-flight limit is reached, which is constantly adjusted to match the consumption rate of the *Shuffle Manager*. Using this approach, the *BlockTransferService* knows in advance what blocks go together and can optimize the construction of the RPC calls.

6 Experimental Evaluation

This section focuses on comparing our proposal with the state-of-the-art to understand the benefits and trade-offs with respect to performance, memory utilization and scalability.

6.1 Experimental setup

We use the following experimental setup throughout our evaluation.

6.1.1 Platform

The experiments were performed on an IBM POWER8 cluster consisting of 14 nodes interconnected with high-speed Mellanox Infiniband EDR ConnectX-4 adapters (100Gb/s). Each node is an IBM POWER8 non-virtualized (PowerNV) S822LC server, featuring two POWER8 SCM (10-core), with 8 simultaneous threads per core (SMT8) for a total of 160 hardware threads, 512 GB of RAM per machine, and one 1 TB HDD for local storage. This gives us a total of 2240 processing units. With respect to the software configuration, the compute infrastructure resources are managed with IBM Platform LSF. Each node runs RedHat Linux Enterprise 7.2, while the Spark version used is 1.6.0. Spark runs on the cluster as an exclusive LSF job (i.e., does not share the node with any other job) and is configured to make use of all available capacity of the nodes. IBM General Parallel File System (GPFS) is used as the underlying distributed storage layer.

6.1.2 Deployment and co-location

Spark was configured to deploy 19 worker instances per node, using a total of 152 threads and 480 GB of RAM from each node. This granularity was chosen in order to enable NUMA awareness: each worker is allocated pre-defined groups of hardware threads by pinning them on the same core, thus maximizing performance for our architecture. We decided to leave 8 hardware threads and 30 GB of RAM free in order to deal with operating system noise. One node acts as the master, while all other nodes host the workers. Each worker runs a single executor. Thus, the largest setup has 1946 reducers with 8 reducers per executor sharing the same initial shuffle blocks. Each reducer needs to fetch additional shuffle blocks from up to 18 other node-local executors (local workers) and 246 remote executors (remote workers).

6.1.3 Approaches

We compare three approaches throughout the evaluation: (1) our proposal (described in Section 4); (2) the default shuffle strategy implemented in Spark (described in Section 5); (3) a naive limitless strategy that issues all requests to all nodes in advance and accumulates all shuffle blocks as they arrive. For the rest of this paper, we refer to our proposal as *adaptive* and to the default Spark shuffle strategy as *default*. Furthermore, in both cases we use a predefined reducer in-flight limit that is configured using the spark.reducer.maxSizeInFlight option. For conciseness, we refer to any strategy and in-flight limit pair simply as *adaptive-N*, and, respectively, *default-N*, with N expressed in Megabytes. The naive limitless strategy is achieved by simply using a very large N for *default*, which forces the desired behavior. Its purpose is to act as a baseline from a theoretical perspective, where memory would not be a limitation and shuffle blocks can freely accumulate indefinitely. We refer to the naive strategy as *unlimited* or *default-unlim*.

6.1.4 Metrics

We are interested to measure the end-impact of the data shuffling strategies on the performance and the memory utilization of the applications. With respect to the performance, we focus on *completion time*, which is what the end-user perceives. With respect to memory utilization, we need to isolate the utilization incurred by the shuffle block accumulation from the rest of the memory management. This is necessary because of the added complexity introduced by the runtime. For example, due to automated JVM garbage collection, the OS-level reported memory utilization does not accurately reflect the active memory utilization. To this end, we instrumented Spark with additional logging mechanisms to capture the moment when a remote shuffle block is received from another node and when it is consumed by the reduce task. By aggregating this information from all nodes, we define the *peak shuffle utilization* as the maximum memory occupied by all unconsumed shuffle blocks throughout the runtime of the application at any moment in time. Finally, we are also interested in system-level metrics such as the evolution in time of average total CPU utilization and network traffic (i.e. average of aggregated values from all nodes at each moment in time), which enables us to correlate the observed low-level behavior with the data shuffling.

TABLE 1
Workload configuration for maximum size

Workload	Parameters	Part.	Red.	Shuffle I/O
groupByKey	KVPairs: 518000 VSize=1000	19760	3952	1 TB
sortByKey	KVPairs: 650000 VSize=1000	18240	3648	1.4 TB

6.1.5 Workloads

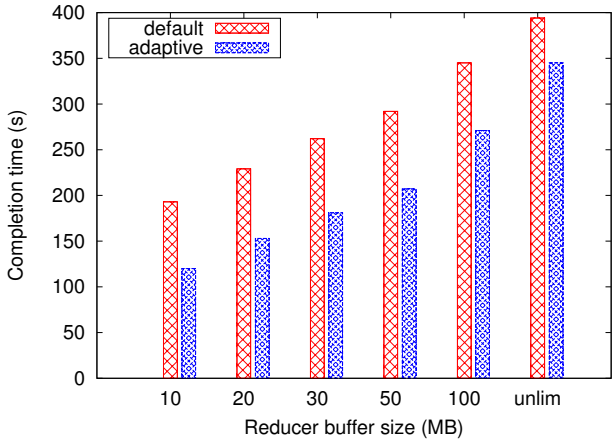
For the purpose of this work, we use two representative patterns that are known to be shuffle-intensive: *groupByKey* and, respectively, *sortByKey*. The former is illustrated by the *GroupByTest* benchmark, which is part of the standard Spark distribution. We extended this benchmark to enable *sortByKey* functionality as well. We opted for this approach instead of using a more specialized sort benchmark (e.g. *TeraSort*) because we wanted to measure the raw power of the Spark *sortByKey* in general, as opposed to specific shuffle strategies customized for the problem (e.g. external shuffle in TeraSort). The *GroupByTest* benchmark includes a data generator that produces a large set of pre-partitioned, pseudo-random key-value pairs. We assigned a predefined seed during the data generation process in order to make sure that the same data will be generated for the same configuration across different runs. The completion time excludes the data generation phase.

6.2 Performance vs. memory utilization

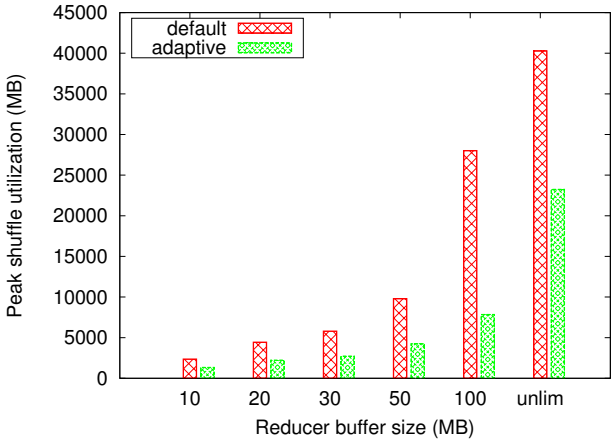
The first series of experiments explores the impact of the shuffle strategy on the performance of the workloads under different reducer in-flight limits. We use a large problem instance that stresses all executors to the maximum capacity. We used 13 worker nodes for *groupByKey* and 12 worker nodes for *sortByKey*. The settings are summarized in Table 1.

For the in-flight limit we use both small realistic sizes (10 MB to 50 MB), one large size (100 MB), and one excessively large size (200 MB) that corresponds to the unlimited case. The default setting is 50 MB.

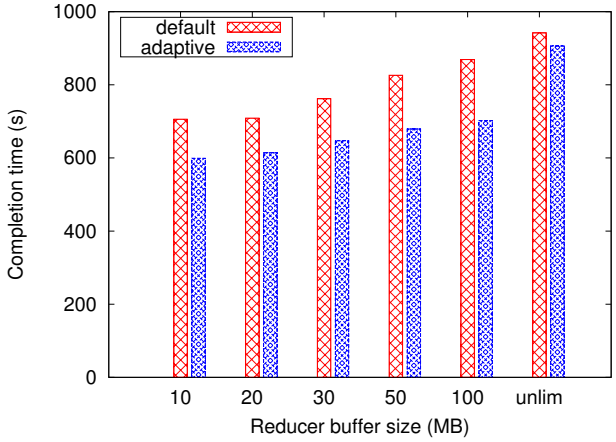
The results are depicted in Figure 3. Performance-wise, it can be observed in Figure 3(a) and Figure 3(c) that for



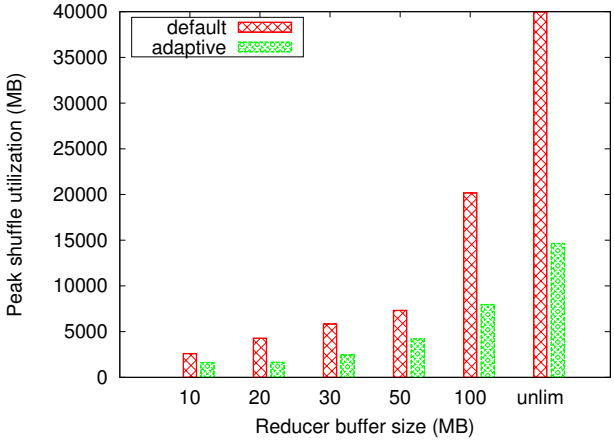
(a) *groupByKey*: Completion time for a variable reducer in-flight limit



(b) *groupByKey*: Peak shuffle memory utilization for a variable reducer in-flight limit



(c) *sortByKey*: Completion time for a variable reducer in-flight limit



(d) *sortByKey*: Peak shuffle memory utilization for a variable reducer in-flight limit

Fig. 3. Performance vs. memory utilization for the largest problem size: 1976 cores for *groupByKey*, 1824 cores for *sortByKey*.

both *adaptive* and *default* there is a significant increase in completion time as the in-flight limit is growing. This is an important observation that contradicts the intuitive expectation of obtaining better performance when larger buffers are available for overlapped communication/computation, since a larger buffer increases the chance of surviving underflows. The explanation for this observation lies in the fact that a large number of reducers per node that are allocated each a large in-flight limit interfere with the rest of the memory management during peaks (e.g. triggers garbage collection more frequently due to memory pressure), which negatively impacts performance. Furthermore, more parallel transfers are happening in the system for a larger in-flight limit, which complicates the bookkeeping (and overhead) needed to keep track of the accumulation of unconsumed shuffle blocks. Thus, it is important to use a low in-flight limit even when memory utilization is not a concern.

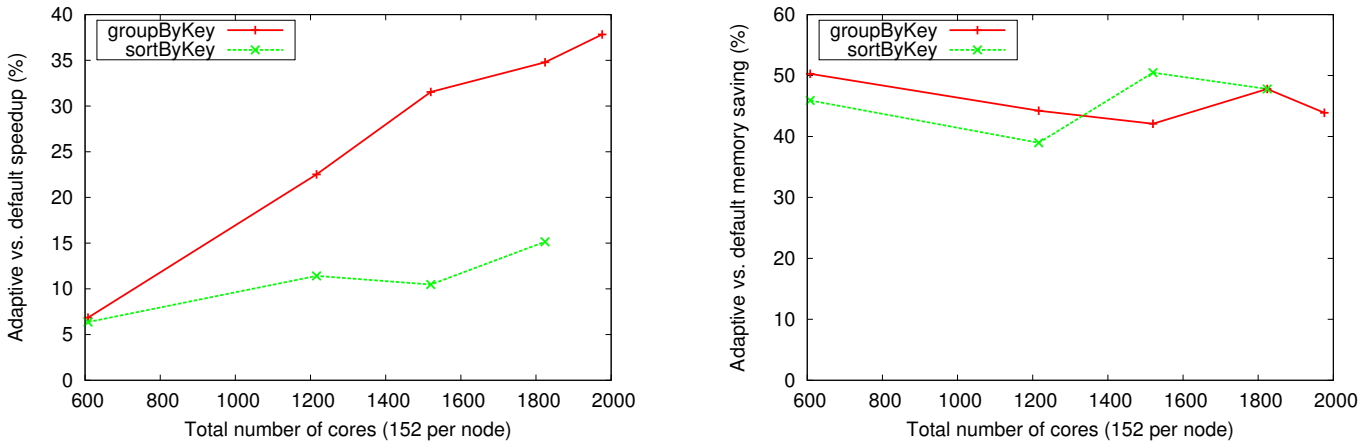
Nevertheless, *adaptive* outperforms *default* consistently regardless of in-flight limit, ranging from up to 20% for *sortByKey* to 40% for *groupByKey*. Also interesting to observe is that the gap between *adaptive* and *default* becomes smaller only in the unlimited case.

A study of the memory efficiency is depicted in Figure 3(b) and Figure 3(d), where the focus is the peak shuffle utilization. As expected, the memory utilization explodes with increasing in-flight limit for both approaches. Interesting to note is that for each in-flight limit, *adaptive* shows a significantly lower memory utilization when compared with *default* (2x less memory for 10 MB inflight limit up to 3x less for *unlimited*). This holds for both workloads. The explanation for this result can be mainly attributed to the elastic adjustment feature implemented by *adaptive*, as explained in Section 4.2.

When correlating the performance with the memory utilization, an important lesson emerges: for a large number of cores per node and low memory amount per core combined with a fast network, the best performance and lowest memory utilization can be achieved simultaneously by using a small reducer in-flight limit rather than the default setting accepted as best practice.

6.3 Scalability

The second series of experiments studies the weak scalability of both approaches. This study is important in order to



(a) Speedup of completion time for *adaptive* vs. *default*

(b) Memory saving for *adaptive* vs. *default*

Fig. 4. Weak performance scalability and memory efficiency for an increasing number of nodes and problem scale using a reducer in-flight limit of 10 MB. Higher is better.

understand what benefits to expect from *adaptive* vs. *default* at increasing scale. Specifically, the experiment consists in deploying an increasing number of executors that solve an increasing problem scale such that the load per executor remains the same, up to the maximum of 14 nodes. This is different from strong scalability, where the overall problem size remains the same but the number of executors varies. In each configuration, we measure the speed-up of *adaptive* vs. *default* with respect to completion time (Figure 4(a)), as well as relative memory utilization reduction (Figure 4(b)). The reducer in-flight limit is fixed at 10 MB, which yields the best performance and memory utilization for both approaches.

With respect to the speed-up of completion time, a clear upward trend is visible for both workloads. In the case of *groupByKey*, there is a 8x increase in speed-up from 608 to 1976 cores: from 5% up to 40%. In the case of *sortByKey*, the speed-up increases 3x from 5% to 15%. Given this trend, an even wider performance gap between *adaptive* and *default* can be expected at scales beyond 2000 cores. With respect to memory utilization, a consistent trend is visible: regardless of the number of cores, *adaptive* utilizes 40%-50% less memory than *default*. At scale, this memory saving can become a critical asset that can be used for user data rather than buffering.

Interesting to note is that the speed-up increases despite a constant memory saving. A possible explanation for this is that every reducer needs to communicate at larger scale with more executors, which emphasizes the need to introduce an optimized source selection algorithm.

6.4 System-Level CPU and Network Utilization Analysis

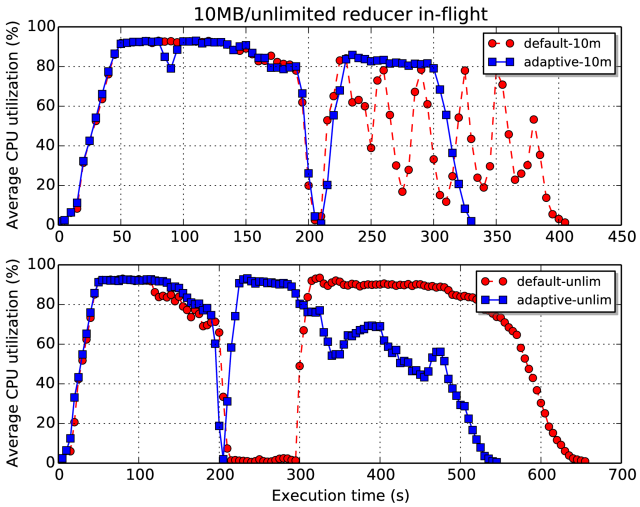
This section aims to study the previous findings by zooming on two low-level system metrics: average aggregated CPU and network utilization (calculated as explained above in Section 6.1.4). We focus on these two metrics to complement the memory utilization that was analyzed so far.

We focus on these two system-level metrics for both workloads at the extreme spectrum of the in-flight reducer limit: *adaptive-10* and the corresponding *default-10*, as well as *adaptive-unlim* vs. *default-unlim* for the largest problem size considered for each of the two benchmarks: 1976 cores for *groupByKey* and 1840 cores for *sortByKey*.

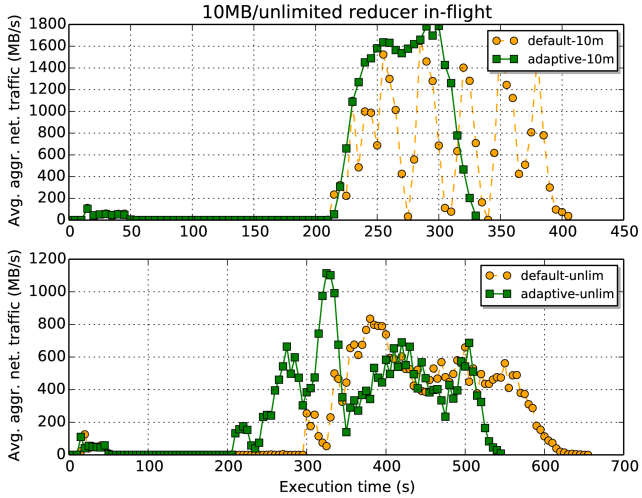
The CPU utilization results are depicted in Figure 5(a) and Figure 5(c), while the network traffic is depicted in Figure 5(b) and Figure 5(d). Note that the depicted system-level parameters include both the data generation phase and the actual workload, as they were monitored throughout the execution time of the benchmark. Since the data generation phase does not involve any shuffling and is identical for both *default* and *adaptive*, the parameters almost overlap until network traffic starts emerging, which signals the beginning of the shuffle phase. It is this part that represents the actual workload and is interesting to analyze.

As can be observed, in the case of *default-10*, large fluctuations in both CPU utilization and network traffic happen over short periods of time for both benchmarks. It indicates the reducers are competing for a large number of shuffle blocks, which creates memory pressure and leads to imbalances, effectively resulting in the reducers "backing off" shortly after. Then, once the pressure is alleviated, the reducers start again competing for shuffle blocks and the cycle repeats. This has negative consequences beyond the fact that it leads to performance degradation and higher memory utilization: these peaks and lows create an unstable pattern that poses difficulties when the infrastructure is shared with other workloads, because it can lead to interference that makes it harder to take appropriate co-scheduling decisions. This is true both if the other workloads are Spark jobs or completely different workloads.

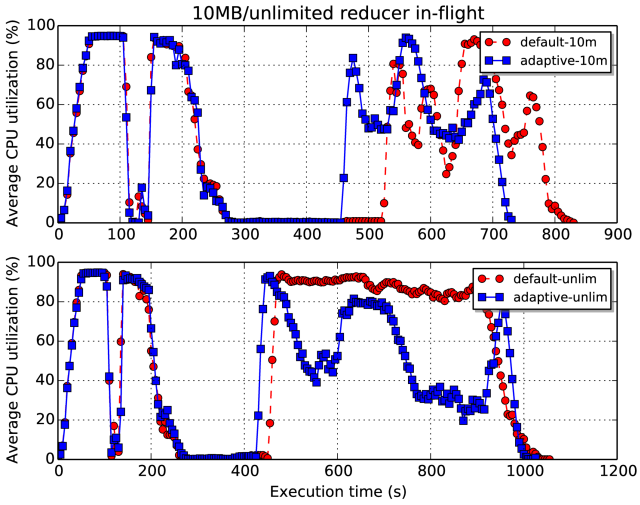
A sharp contrast can be observed when analyzing *adaptive-10* for *groupByKey*: the CPU utilization and network traffic are much more stable over time, which is a consequence of adapting to the computation and improving the load balancing. This in turn helps achieve higher performance and with less memory utilization. In case of *sortByKey*, a better load balancing translates into a noticeably lower CPU utilization. This stability could potentially



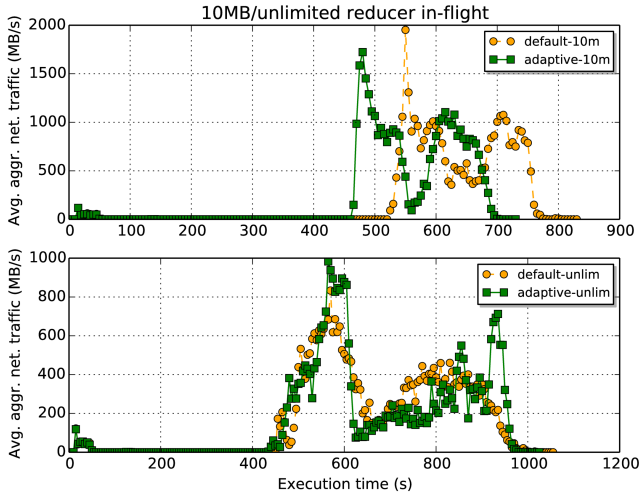
(a) *groupByKey*: average CPU utilization for 10 MB and *unlimited* reducer in-flight limit



(b) *groupByKey*: average network traffic for 10 MB and *unlimited* reducer in-flight limit



(c) *sortByKey*: average CPU utilization for 10 MB and *unlimited* reducer in-flight limit



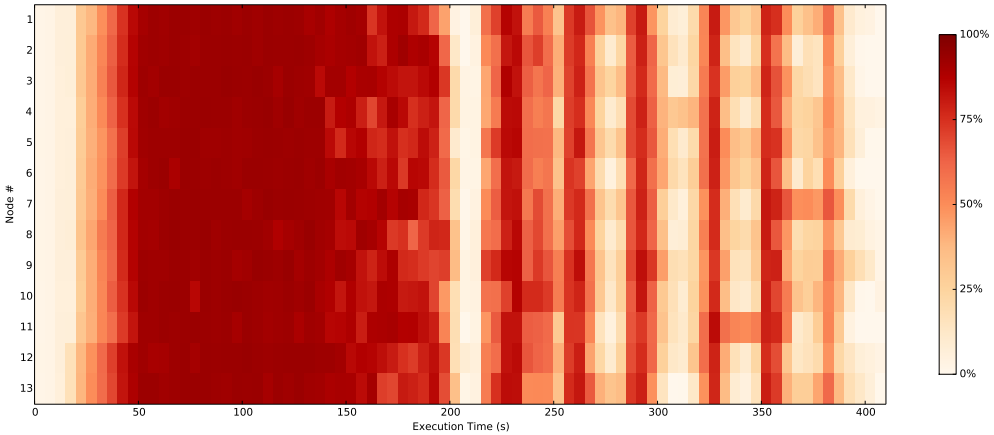
(d) *sortByKey*: average network traffic for 10 MB and *unlimited* reducer in-flight limit

Fig. 5. Average CPU utilization and network traffic per node as they evolve during runtime

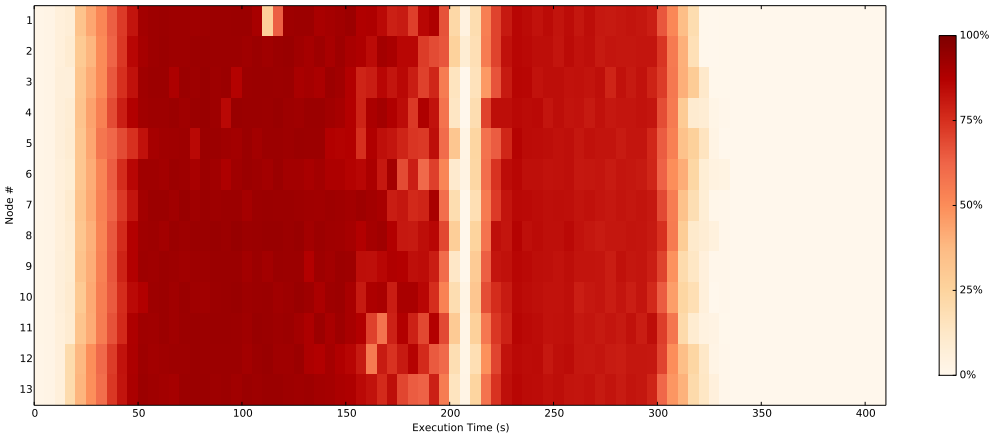
be leveraged by schedulers to make better co-deployment decisions and to improve the quality-of-service. Finally, the network traffic is concentrated in fewer spikes of higher amplitude for both benchmarks, which shows that the interaction of reducers with remote sources is improved and the bandwidth of the network interface can be better utilized.

Since the previous results were based on averages, our next study aims to confirm whether the observed averages are representative of the individual behavior of each node rather than the effect of different behaviors canceling each other out. This question is particularly interesting for the *groupByKey* scenario, where sharp fluctuations of the average CPU utilization are visible. To this end, we investigate this scenario further by creating a *"heatmap"* of the CPU utilization for *default-10* and *adaptive-10*. The result is shown in Figure 6. Specifically, for each node (Y axis) and runtime moment (X axis) the color changes from light (low CPU utilization) to dark (high CPU utilization), as shown in the color scale. For both strategies, it can be observed that the behavior is consistent across all nodes, i.e. the average CPU utilization is representative of the individual CPU utilization of each node. This observation is very important especially in the context of co-scheduling: if all nodes simultaneously exhibit less spikes and more stable behavior for *adaptive*, then schedulers may predict overall behavior easier and could potentially take better global decisions (in addition to local ones at node level).

When comparing *adaptive-unlim* with *default-unlim*, two interesting observations emerge: first, the CPU utilization is visibly smaller for *adaptive-unlim* in both benchmarks during the shuffle phase. Second, the network utilization shows fewer spikes of higher amplitude, similar to the 10 MB inflight limit case. Furthermore, when comparing the 10 MB in-flight limit with the unlimited case, an interesting trend is visible for both benchmarks: the peak network utilization is much higher for the 10 MB inflight limit for



(a) Default strategy using 10 MB reducer in-flight limit



(b) Adaptive strategy using 10 MB reducer in-flight limit

Fig. 6. *GroupByKey*: Heat map of the CPU utilization for all nodes as it evolves during runtime. The adaptive strategy consolidates CPU utilization uniformly over a shorter time compared with the default strategy, which leads to better performance

both benchmarks. The difference can be as high as 2x higher network throughput in the case of *sortByKey*: from 1 GB/s in the case of unlimited to 2 GB/s in the case of 10 MB in-flight limit. This shows again the complexity of the interplay between buffering, memory pressure and the computation, which leads to the observed counter-intuitive behavior that exhibits lower network throughput for a larger buffer.

7 Conclusions

In this paper we have proposed a novel dynamic data shuffling strategy that is specifically designed to deliver high performance and scalability with minimal memory utilization. Our proposal is based on the idea of adapting the accumulation of shuffle blocks to the individual rate of processing for each reducer task, while coordinating the reducers to collaborate in the optimal selection of the sources where to fetch shuffle blocks from. This improves load balancing and avoids stragglers, while reducing the memory which is needed for buffering purposes.

To demonstrate the benefits of our proposal, we developed an experimental prototype that we integrated into the Spark framework as an alternative data shuffling strategy. We ran extensive experiments on high-end HPC infrastructure with large core count per node and fast interconnect to compare this alternative data shuffling strategy with the default one for two shuffle-intensive benchmarks. Our first key finding is the following: when under memory pressure, the default settings considered best practice are not optimal and a small reducer in-flight limit is necessary to achieve both the best performance and minimum memory utilization. Our second key finding is that our proposal exhibits increasing speed-up at scale vs. the default strategy when using an small reducer in-flight limit: from 5% for both benchmarks at around 600 cores up to 15% and, respectively, 40% at 2000 cores. This translates to an increase in speed-up of 8x and, respectively 3x, for a corresponding 3x increase in the number of cores.

Given this trend, we predict even higher speed-up at larger scale. Furthermore, the speed-up comes with the added benefit of halving the memory requirement for the buffering of accumulated shuffle blocks. Since less and less memory per core is available, due to an increase in number of cores per node, this translates to significant savings at scale. We explained these findings by analyzing CPU and network utilization, showing more consistent use of both resources. This more stable usage of CPU and networking infrastructure is also interesting because it creates opportunities for less interference and better co-location with other workloads.

7.1 Future work

Encouraged by these promising results, we see several interesting avenues to be explored in future work. 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 to 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 CPU utilization and network transfers. 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] European Exascale Software Initiative, "D4.3 Final Report on Enabling Technologies," 2015, EESI2 312478.

[5] B. Nicolae, C. Costa, C. Misale, K. Katrinis, and Y. Park, "Towards memory-optimized data shuffling patterns for big data analytics," in *CCGrid'16: 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Cartagena, Colombia, 2016, pp. 409–412, short Paper.

[6] 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.

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

[8] 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.

[9] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou, "Workload characterization on a production hadoop cluster: A case study on taobao," in *IISWC '12: Proceedings of the 2012 IEEE International Symposium on Workload Characterization*. San Diego, USA: IEEE Computer Society, 2012, pp. 3–13.

[10] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality and fairness in mapreduce," in *MapReduce '12: The Third International Workshop on MapReduce and Its Applications*. Delft, The Netherlands: ACM, 2012, pp. 25–32.

[11] 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.

[12] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou, "Optimizing Data Shuffling in Data-parallel Computation by Understanding User-defined Functions," in *NSDI'12: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, USA, 2012, pp. 22:1–22:14.

[13] 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.

[14] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "Mronline: Mapreduce online performance tuning," in *HPDC '14: The 23rd International Symposium on High-performance Parallel and Distributed Computing*. Vancouver, Canada: ACM, 2014, pp. 165–176.

[15] D. Cheng, J. Rao, Y. Guo, and X. Zhou, "Improving mapreduce performance in heterogeneous environments with adaptive task tuning," in *Middleware '14: The 15th International Middleware Conference*. Bordeaux, France: ACM, 2014, pp. 97–108.

[16] C. L. Abad, N. Roberts, Y. Lu, and R. H. Campbell, "A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns," in *IISWC '12 Proceedings of the 2012 IEEE International Symposium on Workload Characterization*, San Diego, USA, 2012, pp. 100–109.

[17] C. L. Abad, H. Luu, N. Roberts, K. Lee, Y. Lu, and R. H. Campbell, "Metadata traces and workload models for evaluating big storage systems," in *UCC '12: Proceedings of the 5hth International Conference on Utility and Cloud Computing*. Chicago, USA: IEEE Computer Society, 2012, pp. 125–132.

[18] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, pp. 169–184, February 2011.

[19] 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.

[20] N. S. Islam, X. Lu, M. W.-u. Rahman, and D. K. D. Panda, "Sor-hdfs: A seda-based approach to maximize overlapping in rdma-enhanced hdfs," in *HPDC '14: The 23rd International Symposium on High-performance Parallel and Distributed Computing*. Vancouver, BC, Canada: ACM, 2014, pp. 261–264.

[21] F. J. Clemente-Castelló, B. Nicolae, K. Katrinis, M. M. Rafique, R. Mayo, J. C. Fernández, and D. Loreti, "Enabling Big Data Analytics in the Hybrid Cloud Using Iterative MapReduce," in *UCC'15: 8th IEEE/ACM International Conference on Utility and Cloud Computing*, Limassol, Cyprus, 2015, pp. 290–299.

[22] B. Nicolae, P. Riteau, and K. Keahey, "Towards Transparent Throughput Elasticity for IaaS Cloud Storage: Exploring the Benefits of Adaptive Block-Level Caching," *International Journal of Distributed Systems and Technologies*, vol. 6, no. 4, pp. 21–44, 2015.

[23] 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.

[24] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng, "HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment," in *CLUSTER'09: IEEE 2009 International Conference on Cluster Computing and Workshops*, New Orleans, USA, 2009, pp. 1–8.

[25] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman, "Numa-aware algorithms: the case of data shuffling," in *CIDR '13: The 6th Biennial Conference on Innovative Data Systems Research*, Asilomar, USA, 2013.

[26] 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.

[27] 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.

[28] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 98–109, 2011.

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

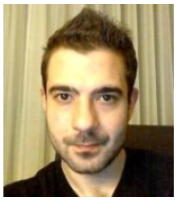
[30] B. Nicolae, "On the benefits of transparent compression for cost-effective cloud data storage," *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, vol. 3, pp. 167–184, 2011.

[31] ——, "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.

Bogdan Nicolae is a research scientist within the High Performance Systems group at IBM Ireland. He specializes in scalable storage and fault tolerance for large scale distributed systems, with a focus on cloud computing and high performance architectures. He holds a PhD from University of Rennes 1, France and a Dipl. Eng. degree from Politehnica University Bucharest, Romania. He is interested by and authored numerous papers in the areas of scalable I/O, storage elasticity and virtualization, data and metadata decentralization and availability, multi-versioning, checkpoint-restart, live migration. He is also actively involved in the organization of international conferences (e.g., SC, IPDPS, HPDC, PPoPP, CCGrid, CLUSTER, CLOUD) and editing of journals (e.g., IEEE TCC) relevant to these areas.

Carlos H. A. Costa is a Research Staff Member at IBM T. J. Watson Research Center, where he is part of the System Software group within the Data-Centric Systems (DCS) department. His research is mainly focused on system software, programming models and middleware for next-generation HPC systems, working at the intersection of traditional HPC and emerging large-scale analytics frameworks and workloads. His work also includes novel methods for software/hardware cooperation to improve system resiliency. His areas of expertise are high-performance computing, system software, fault-tolerance, code generation and runtime optimization, energy-aware computing and networking. He received his Ph.D and M.Sc. degrees in computer engineering from University of Sao Paulo (Brazil) and his B.Sc. degree in computational physics from University of Brasilia (Brazil).

Claudia Misale is a PhD candidate at Computer Science Department of the University of Torino and a member of the parallel computing Alpha group. She was participating in the European STREP FP7 ParaPhrase and REPARA projects and she has been a research intern in the High Performance System Software research group in the in the Data Centric Systems Department at IBM T.J. Watson, working on optimizing big data analytics frameworks for Data-Centric Systems (DCS). Her research is focused on high performance computing, in particular on high level models and patterns for distributed computing and high-performance big data analytics for HPC platforms.



Kostas Katrinis is interested in various aspects around computing as a service (be it virtualized or bare-metal) and data-centric computing, which are driven by the semantics and system utilization profile of key applications/workloads of high business/scientific value. He is interested (among other things) by: technology foundations and platforms for in-memory analytics, convergence of data-centric and high performance computing systems, software-defined cloud computing. He specializes in the area of communication/interconnection networks, from architecture/protocol design to resource dimensioning to systems/protocols performance analysis and optimization. He holds a PhD in Technical Sciences from the ETH Zurich, Switzerland and a Diploma in Computer Engineering and Informatics from the University of Patras, Greece.

Yoonho Park is a Research Staff Member at IBM T. J. Watson Research Center, and he manages the System Software Team in the Data Centric Systems Department. His research activities include high-performance computing, operating systems, and middleware. He is a Senior Member of the ACM and a member of USENIX. He received his Ph.D. from the University of Michigan in 1997.