An RDMA Middleware for Asynchronous Multi-stage Shuffling in Analytical Processing

Rui C. Gonçalves[1(✉)], José Pereira[1], and Ricardo Jiménez-Peris[2]

[1] HASLab, INESC TEC & U. Minho, Braga, Portugal
{rgoncalves,jop}@di.uminho.pt
[2] Univ. Politécnica de Madrid & LeanXcale, Madrid, Spain
rjimenez@leanxcale.com

Abstract. A key component in large scale distributed analytical processing is *shuffling*, the distribution of data to multiple nodes such that the computation can be done in parallel. In this paper we describe the design and implementation of a communication middleware to support data shuffling for executing multi-stage analytical processing operations in parallel. The middleware relies on RDMA (Remote Direct Memory Access) to provide basic operations to asynchronously exchange data among multiple machines. Experimental results show that the RDMA-based middleware developed can provide a 75 % reduction of the costs of communication operations on parallel analytical processing tasks, when compared with a sockets middleware.

Keywords: Distributed databases · OLAP · Middleware · RDMA

1 Introduction

The proliferation of web platforms supporting user generated content and of a variety of connected devices, together with the decreasing cost of storage, lead to a significant growth on data being generated and collected every day. This explosion of data brings new opportunities for businesses that overcome the challenge of storing and processing it in a scalable and cost-effective way. It has thus sparked the emergence of NoSQL database systems and processing solutions based on the MapReduce [2] programming model as alternatives to the traditional Relational Database Management Systems (RDBMS) for large scale data processing.

Briefly, in a MapReduce job, a *map* function converts arbitrary input data to key-value pairs. For instance, in the classical word count example, for each input text file, *map* outputs each word found as a key and its number of occurrences as the value. A *reduce* function computes an output value from all values attached to the same key. For instance, in the word count example, *reduce* sums all values for each key to obtain the global count for each word. Both these operations can easily be executed in parallel across a large number of servers with minimal coordination: Multiple mappers work on different input files, and multiple reducers work on different keys.

M. Jelasity and E. Kalyvianaki (Eds.): DAIS 2016, LNCS 9687, pp. 61–74, 2016.
DOI: 10.1007/978-3-319-39577-7_5

The key element of a MapReduce implementation, which needs distributed coordination, is the *shuffling* step between map and reduce operations. It gathers all data elements with the same key on the same server such that they can be processed together. In classical MapReduce, this is a synchronous step: All map tasks have to finish before reduce tasks can be started. This impacts the latency of MapReduce jobs, in particular as multiple map and reduce stages are often needed to perform data processing operations. Therefore, it restricts the usefulness of systems based on MapReduce to batch processing, even if, as in Hive [15], they offer a high-level SQL-like interface.

There has thus been a growing demand for NoSQL solutions that combine the scalability of MapReduce with the interactive performance of traditional RDBMS for on-line analytical processing (OLAP). For instance, Impala [6] offers the same interface as Hive but avoids MapReduce to improve interactive performance. Again, a key element in these data processing systems is the ability to perform shuffling efficiently over the network. In detail, the shuffling has to be asynchronous, to allow successive data processing tasks to execute in parallel, and multi-stage, to allow an arbitrarily long composition of individual tasks in a complex data processing job. Being the component that involves distributed communication and synchronization, shuffling is the key component for the performance and scalability of the system.

This paper presents an asynchronous and multi-stage shuffling implementation that exploits the Remote Direct Memory Access (RDMA) networking interface to add analytical processing capabilities to an existing Distributed Query Engine (DQE) [7]. The DQE provides a standard SQL interface and full transactional support, while scaling to hundreds of nodes. Whereas the scalability for on-line transactional processing (OLTP) workloads is obtained executing multiple transactions (typically short lived) concurrently on multiple DQE instances, for OLAP workloads it is also important to have multiple machines and DQE instances computing a single query in parallel. That is, as OLAP queries have longer response times, it is often worth considering intra-query parallelism to reduce queries response time [8].

The parallel implementation of the DQE for OLAP queries follows the *single program multiple data* (SPMD) [1] model, where multiple symmetric *workers* (threads) on different DQE instances execute the same query, but each of them deals with different portions of the data. The parallelization of stateful operators requires shuffling rows, so that the same worker processes the related rows. Shuffling is done using a communication middleware that provides all-to-all asynchronous data transfers, which was initially implemented using non-blocking Java sockets. In this paper we describe an RDMA-based implementation of the middleware, which was developed as an alternative to reduce the communication overheads associated with parallel execution of OLAP queries, and we discuss aspects considered while redesigning the middleware to leverage from RDMA technologies.

Our middleware implementation relies on the RDMA Verbs programming interface, and uses one-sided write operations for data transfers, and send/receive

operations for coordination messages. For improved performance, it makes heavy use of pre-allocated and lock-free data structures to operate. Moreover, it uses batching to make a more efficient use of network. Experimental results show that our RDMA-based middleware implementation can provide a 75 % reduction on communication costs, when compared with a sockets implementation.

The rest of this paper is structured as follows: In Sect. 2, we describe the requirements for supporting shuffling and the functionality offered by RDMA networking. Section 3 describes the proposed solution. Section 4 compares the proposed solution to a sockets-based middleware and Sect. 5 contrasts it to alternative proposals. Finally, Sect. 6 concludes the paper.

2 Background

2.1 Shuffling

In the DQE, shuffle operators are used when parallelizing stateful operators to redirect rows to a certain worker based on a hash-code (computed from the attributes used as key by the stateful operator being parallelized). Shuffle operators are also used to redirect all results to the master worker at the end of the query, or to broadcast rows from sub-queries.

The communication middleware provides efficient intra-query synchronization and data exchange, and it is mainly used for exchanging rows in shuffle operators. A push-based approach is followed. When processing a row that should be handled by other worker, the sender immediately tries to transfer it. Each receiver maintains *shuffle queues* (Fig. 1), which are used to asynchronously receive the rows. The shuffle queues abstract a set of queues used by a worker to receive rows from the other workers, and they contain an incoming and an outgoing buffer per each other worker, which are used to temporarily store rows being exchanged. That is, the rows are initially serialized to the appropriate outgoing buffer (on the sender side), and then the serialized data is transferred to the matching incoming buffer of the receiver worker, using the communication middleware. An optimization is made for the case where the receiver worker is running on the same DQE instance of the sender. In those cases, the shared session state is used to allow the sender to directly move the rows to the shuffle queues of the receiver.

Multiple shuffle operators may be required by a parallel query plan, thus the need for multi-stage shuffling. To reduce the memory cost associated to buffers – which increases quadratically with the number of workers – there is a single incoming and a single outgoing buffer shared by all shuffle operators (multiplexing is used to logically separate data from multiple shuffle operators).

The communication middleware was initially implemented using Java sockets. For this implementation, a communication end-point is created when initializing a worker, which means to start a server socket and bind it to the IP address of the machine. Then a non-blocking socket channel is opened between each pair of workers running on different DQE instances, and the associated incoming/outgoing buffers are allocated.

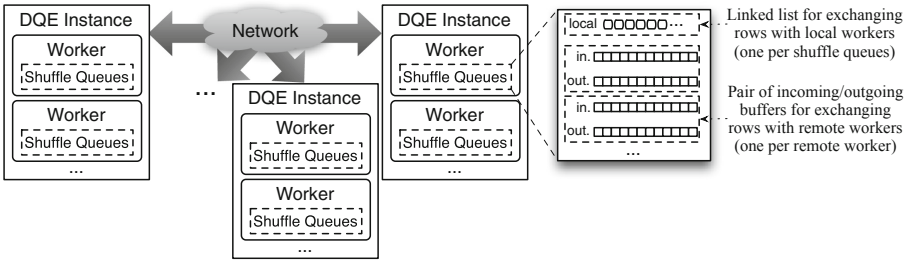


Fig. 1. DQE architecture and shuffle queues structure.

When a row is requested by a shuffle operator, the operator starts by polling its shuffle queues, where it may have received rows from other workers. The polling process of shuffle queues comprises the following steps:

- Check if there is a row received from a worker from the same DQE instance.
- If no row is available:
  - Read (copy) data available on socket channels to incoming buffers.
  - Poll the incoming buffers for available rows for the current shuffle operator.

If a row is obtained, it is returned by the shuffle operator. However, polling shuffle queues may return no rows. In that case, the shuffle operator obtains a local row from its child task/operator (as defined in the query plan). The row is hashed to determine the worker that should process it. If it is a row for the current worker, it is returned by the shuffle operator. Otherwise it is sent to the appropriate worker, which implies serializing the row to an outgoing buffer, and writing the data available to the socket channel. As the shuffle operator still does not have a row to return, it goes back to the polling process and it tries again to obtain a row for itself. As long as the shuffle operator has local rows to process from its child operator, it does not block polling the shuffle queues. After processing all those rows, the worker blocks if polling the shuffle queues returns no rows. It will poll the shuffle queues again as soon as new data is received. The only other situation where the worker may block is when there is no free space on an outgoing buffer when sending a row to a remote worker.

In summary, the push-based asynchronous shuffling approach followed by the DQE requires the following key functionalities from the communication middleware [5]: ability to send and queue rows on remote workers; ability to retrieve the rows queued; ability to block a worker when there are no rows to process (and to wake it up when new rows are received); and ability to block a worker when a row cannot be immediately copied to a buffer (and to wake it up when space becomes available).

2.2 RDMA Verbs

RDMA protocols [12] provide efficient and reliable mechanisms to read/write data directly from the main memory of remote machines, without the involvement of the remote machine CPU, enabling data transfers with lower latency

and higher throughput. By providing applications with direct access to network hardware, RDMA also bypasses typical complex network stacks and operating system, reducing memory copies and CPU usage. The RDMA Verbs is the basic programming interface to use RDMA protocols, and it supports data transfers using either one-sided read/write operations, or two-sided send/receive operations. Additionally, there is the *write with immediate data* operation, which is a one-sided write operation that also triggers a receive operation.

The API is asynchronous, that is, *queue pairs* – comprised of a send and a receive queue – are used to queue operation requests for each connection. The application may choose to receive *completion events* when requests are finished, which are posted into a *completion queue* associated with the queue pair. To avoid active polling, the application may request to be notified when a completion event is added to the completion queue (these notifications are sent to a *completion channel*).

In our work, we used the jVerbs library [13], a Java implementation of the RDMA Verbs interface available on the IBM JRE. Besides providing an RDMA Verbs API for Java, jVerbs relies on modifications of the IBM JVM to reduce memory copies.

3 Middleware Design and Implementation

In this section we first present the design of the RDMA communication middleware developed, and we then discuss in detail the implementation decisions critical to the performance of our solution.

3.1 Design Overview

The RDMA middleware relies on one-sided RDMA write operations to transfer rows' data directly between Java memory buffers, and send/receive operations for coordination.

When initializing workers for a parallel connection, on each DQE instance running workers, an RDMA server connection is created and bound to the machine IP address. Then all DQEs are connected with each other, which requires (i) to pre-allocate and initialize memory buffers, queue pairs, completion channel, and completion queue; (ii) to start a new thread (the network thread), which will handle the work completion events; (iii) to start RDMA connections with all other DQE instances; and (iv) to pre-allocate and initialize the data structures needed to execute the network requests.

These steps are performed when opening a database connection, where it is specified the level of parallelism – number of workers to use – for queries executed using that connection. In this way, the overheads of preparing the network are avoided during the execution of queries. On the other hand, the resources remain allocated even if the connection is not being used to run queries.

As described before, when executing a shuffle operator, workers send and receive rows asynchronously through shuffle queues, which use buffers to serialize

and temporarily store those rows until they are polled on the receiving side. However, when using the RDMA middleware, the sender uses an RDMA write request to transfer the serialized data from one of its outgoing buffers to a remote incoming buffer. Then, after the network thread receives a work completion event confirming the execution of the RDMA write request, the receiving side is notified, and the tail of the local outgoing buffer is updated, to release the space occupied by the data transferred during the request. The sending side takes into account the tail position of the remote buffer to determine the free space available. When there is no space available on the remote buffer, the data transfer can only occur after the network thread receives a notification updating the tail of the remote buffer (i.e., releasing space on the remote buffer), thus the network thread assumes the task of posting the RDMA write request, and the worker proceeds with its operation, unless the local outgoing buffer is also full. In this case, instead of spilling data to disk – as it is done in some MapReduce implementations, for example –, we chose to block the worker, until space is released.

When workers want a new row to consume, they follow the polling process described in Sect. 2.1. However, as now data is transferred using RDMA write operations, some changes are required. Firstly, the workers do not have to copy data from the channels to their incoming buffers, as the data is transferred directly to those buffers. Moreover, as the data is transferred without the intervention of the receiving side, the network thread uses the notifications previously described to keep track of buffers with data available for each worker, and it wakes up blocked workers when it receives notifications.

3.2 Implementation Decisions

Network and Worker Threads. We use a thread dedicated to track completion of operations (the network thread). To reduce CPU consumption, this thread blocks waiting for completion events, and it is in charge of operations that follow a completion event of a network operation. This includes to process the completion of RDMA write requests (sending the needed notifications, and updating outgoing buffer states), as well as processing received notifications (possibly waking up blocked worker threads). As this thread blocks waiting for completion events, we decided to not use this thread to post the RDMA write requests, as the requests would not be posted until the network thread wakes up. Worker threads are in charge of performing the RDMA write requests to transfer rows, with one exception: In case there is an ongoing RDMA write request, the new request is delayed until the previous one completes. As it is the network thread that tracks the completion of the requests, it is also this thread that will post the RDMA write requests in those cases. As after returning from the sending operation workers may want to reuse the memory space that contains the row to send, the sending operation always serializes the row to the outgoing buffer (even if it does not perform the RDMA write request). Therefore, if this buffer is full the worker blocks. The alternative would imply to copy the row to a temporary buffer, or to spill data to disk, as we mentioned previously. As typically there are

many other threads to keep the system busy, we choose this option that avoids wasting CPU time.

RDMA Connections. A single connection/queue pair is used per pair of machines, which means that multiple workers share the same connection/queue pair. In this way, if we have m machines with n workers each, we require $m - 1$ connections per machine. If we used a connection for each pair of workers, we would require $n \times n \times (m - 1)$ connections per machine (i.e., for each of the n workers on a machine, there would be a connection to each of the n workers on every other $m - 1$ machines). We followed this approach to reduce the needs of on-chip memory of the network card, which can compromise the scalability of the communications [3]. Regarding memory buffers, we use a single contiguous memory region per pair of machines, which is later divided in multiple buffers, to be used by the different pairs of workers.

Notifications. To detect the availability of new received data, we decided to use send/receive requests to notify the receiving side. The main goal was to avoid active polling on all incoming buffers, which results in scalability problems. As receivers are notified when data is written/received, they can easily keep track of the list of buffers with data available. An alternative would be to use an *RDMA write with immediate data*, but this operation is not provided by the jVerbs API. Moreover, the notifications are also used to notify the sending side that data was read from a buffer, which is essential to determine when data can be transferred. To reduce the number of read notifications, they are only sent after reading a configurable amount of data (an approach similar to the one followed by [3]). That is, the sender does not have knowledge of the released space immediately. Although this could make workers block more often when sending rows, our experiments showed that workers rarely block in these situations.

Batching. In the initial implementation, the middleware was prepared to transfer data as soon as it was available, in order to reduce latency. However, due to the small size of the rows being transferred, we noticed that this could result in significant communication overheads, particularly when using an RDMA software implementation such as Soft-iWARP [17]. Due to the asynchronous nature of the DQE, the latency is not critical. Therefore, the middleware provides the ability to define a minimum threshold of data, that is, the data transfer request is delayed until a certain amount of data to transfer is available (or a *flush* operation is performed). This threshold may be adjusted, namely to take into account the network hardware characteristics (i.e., we can use lower thresholds when using network hardware with support for RDMA). Moreover, notifications are also sent in batches. That is, when performing actions that originate multiple notifications, the notifications are initially queued, and at the end they are sent in a batch.

Lock-free Pre-initialized Data-structures. For increased performance, the middleware makes use of lock-free data structures, allowing worker threads to operate without blocking, until they have no work to process. The network thread

blocks waiting for completion events, but the middleware is designed so that worker threads are not prevented from progress in this case. The incoming and outgoing buffers are implemented using *circular buffers* on top of direct byte buffers (i.e., this memory is outside of the Java garbage-collected heap). These circular buffers are designed to support a write and a read operation concurrently without using locks, to avoid contention when serializing rows. Moreover, the main data structures needed are initialized during connection, and are reused for all queries executed with the connection. To reduce overheads associated to JNI serialization when jVerbs makes RDMA verbs calls to lower level libraries, jVerbs provides *stateful verbs methods (SVM)*, which cache the serialized state of the call, enabling this state to be reused in later calls without additional serialization overheads. By making use of this mechanism, and by initializing the SVM objects during connection, we keep these overheads outside the execution of queries.

RDMA Writes vs Send/Receive. We decided to use RDMA writes to transfer data. Regarding performance, RDMA write requests usually provide better latencies and lower CPU usage on the passive side [11]. Even though the latency is not critical, the lower CPU usage is important to leave more resources for the worker threads. Moreover, RDMA write requests also simplify the communication process, as a single connection is used to transfer data between multiple pairs of buffers. That is, the receiver does not know in advance where the received data should be placed. Whereas with RDMA write operations it is the sender that determines where the data is placed on the receiving side, with send/receive operations this is determined by the receiver. Therefore, to use send/receive requests the sender would need to tell the receiver in advance the buffer to use to receive the data. The receiver would then post a receive request with the appropriate buffer, and tell the sender it could send the actual data (or tell the sender it cannot send data if there is no buffer space on the receiving side). This increases the number of requests to transfer data, and it forces the data transfer operations to be posted one at a time, to make sure that data is placed on the right buffer on the receiving side, whereas our current solution allows for multiple posted RDMA write requests pending completion. To avoid this, we would have to either use a single buffer per pair of DQE instances (instead of a single buffer per pair of workers), or an RDMA connection per pair of workers. The former solution would impose contention among workers when serializing and deserializing rows. The latter would increase the number of connections needed, which would compromise scalability, as we discussed previously in this section.

4 Evaluation

To evaluate the solution developed we conducted performance experiments, which we report in this section. First we compare the RDMA middleware with the original sockets middleware in a synthetic benchmark, which simulates the use of the middleware to execute queries, but that removes all the computation

related with the actual query execution, leaving only the shuffle operators. Then we compare both middleware implementations executing real analytical queries with the DQE.

The evaluation was conducted using a cluster of 9 servers. All servers have Intel Core i3 CPUs, with 2 physical cores (4 threads), 8GB of RAM, SATA HDD (7200 rpm), and GigaBit Ethernet. As the servers do not have network hardware supporting RDMA, we used Soft-iWARP [17].

4.1 Synthetic Benchmark

In this section we compare the performance of the middleware implementations using an application that simulates the execution of shuffle operators in real queries, but without operators that do the actual query computation. That is, each worker thread of the application executes a "query plan" that essentially contains two shuffle operators (see Fig. 2). The "rows" are integers generated sequentially (node `Int Generator` on Fig. 2), and between the two shuffle operators a simple transformation is applied to the integers received from the previous operator to make sure that most of them will be sent to a different worker in the next shuffle operator.

For these experiments we used a setup with 4 servers running one application process each, and another setup with 8 servers running one application process each. The tests were conducted using IBM JRE 8.0-1.10. The size of the buffers used by the communication middleware was set to 64 KB (the default value). Each application process generates 5M integers, which are shuffled twice (i.e., the shuffle operators of each process handle 10M integers in total).

We measured the execution time with different numbers of workers on each process, both using the sockets and the RDMA middleware. The execution times (averages of 8 executions) are reported on Fig. 3. Considering the fastest times for each middleware in the two setups tested, we can observe that the RDMA middleware resulted in a reduction of around 75 % of the execution time.

We also used this synthetic application to illustrate the impact of batching multiple rows before transferring them, as described in Sect. 3.2. Figure 4

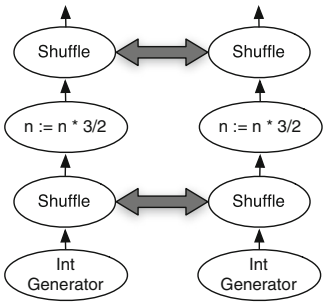
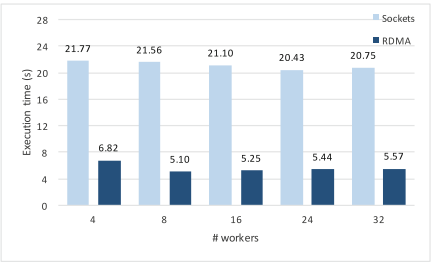


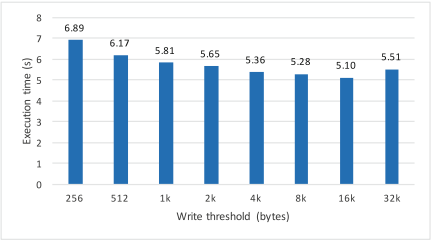
Fig. 2. Plan used for the synthetic benchmark.



(a) 4 processes/machines.

(b) 8 processes/machines.

Fig. 3. Execution times of the synthetic application for the sockets and RDMA middleware, when varying the number of DQE instances and the number of workers.



(a) 4 processes/machines, 2 workers per process.

(b) 8 processes/machines, 2 workers per process.

Fig. 4. Execution times of the synthetic application with different thresholds for write requests batching.

shows the execution times using different minimum thresholds for transferring data, when using 4 processes and 8 processes (in both cases 2 worker threads per process were used). As we can observe, when using the RDMA middleware adjusting this threshold can lead to variations on the execution time higher than 25 %. With the used hardware we observed a good performance with a threshold of 16 KB, which is the value we used in the other tests reported in this section.

4.2 Application Benchmark

We also compared the performance of both middleware implementations using the DQE to run analytical queries. These tests were conducted using 3 analytical queries, executed over a TPC-C database [16]. Listing 1.1 shows the queries used, which expose different combinations of common operators of analytical queries.

For this experiment we used the following setups: 4 servers running DQE instances and the key-value data store component (HBase), and 1 server running the remaining services required by the DQE; and 8 servers running DQE instances and the key-value data store component, and 1 server running the remaining services required by the DQE. The tests were conducted using HBase 0.98.6, running on Oracle JRE 1.7.0_80-b15, and the DQEs were running on IBM

JRE 8.0-1.10. The size of the buffers used by the communication middleware was set to 64 KB (the default value).

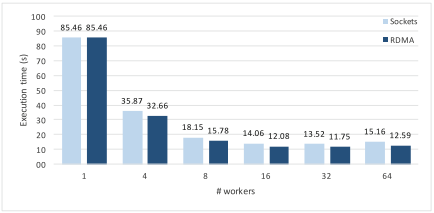
```
-- Query 1
select ol_o_id, ol_w_id, ol_d_id, sum(ol_amount) as revenue, o_entry_d
    from order_line, orders, new_order, customer
    where c_id = o_c_id and c_w_id = o_w_id and c_d_id = o_d_id
        and no_w_id = o_w_id and no_d_id = o_d_id and no_o_id = o_id
        and ol_w_id = o_w_id and ol_d_id = o_d_id and ol_o_id = o_id
        and c_state like 'A%' and o_entry_d > timestamp('2013-07-01-00.00.00.000000')
    group by ol_o_id, ol_w_id, ol_d_id, o_entry_d
    having sum(ol_amount) > 80000.00
    order by revenue desc, o_entry_d;

-- Query 2
select 100.00 * sum(case when i_data like 'a%' then ol_amount else 0 end) /
      (1+sum(ol_amount)) as promo_revenue
    from order_line, item
    where ol_i_id = i_id
        and ol_delivery_d >= timestamp('2013-06-01 00:00:00.000')
        and ol_delivery_d < timestamp('2013-08-01 00:00:00.000');

-- Query 3
select ol_number, sum(ol_quantity) as sum_qty, sum(cast(ol_amount as decimal(10,2))) as
      sum_amount, sum(ol_quantity) / count(*) as avg_qty, sum(ol_amount) / count(*) as
      avg_amount, count(*) as count_order
    from order_line
    where ol_delivery_d > timestamp('2013-07-01 00:00:00.000')
    group by ol_number order by sum(ol_quantity) desc;
```

Listing 1.1. Evaluation Queries.

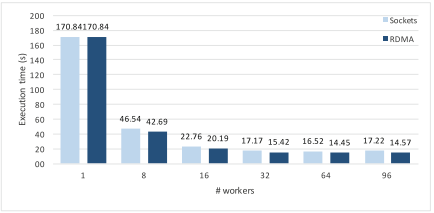
We measured the execution time of the queries previously presented, running both without parallelization and with parallelization (with different numbers of workers), over TPC-C databases with a scale factor of either 15 (for the setup with $4+1$ servers) or 30 (for the setup with $8+1$ servers). The parallel times were obtained both for the sockets middleware and for the RDMA middleware. For each different setup, the queries were run 5 times, and the average of the last 4 runs was considered, to account for cache warm-ups.

Figure 5 shows the executions times obtained for the different setups. As we can observe from the results obtained, when using 4 DQE instances, the RDMA middleware resulted in improvements on the maximum speedup between 2.0 % (from 5.97x to 6.09x, in Query 2) and 16.2 % (from 6.44x to 7.49x, in Query 3), when compared with the sockets middleware. When using 8 DQE instances, the RDMA middleware resulted in improvements on the maximum speedups between 6.1 % (from 13.52x to 14.35x, in Query 3) and 14.3 % (from 10.34x to 11.83x, in Query 1).

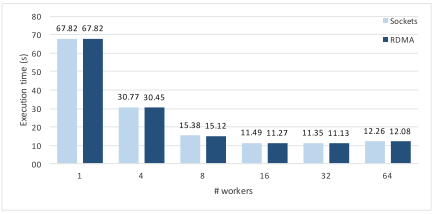
As it would be expectable, significant improvements were obtained with Query 1, which is the query that involves more shuffle operators to be parallelized (thus, more communications). That is, the RDMA middleware is particularly important to the parallelization of more complex queries. On the other hand, Query 2 obtained lower benefits from the RDMA middleware, as it performs less communication. Load balancing issues affecting this query also contribute to the lower benefits obtained from using the RDMA middleware.



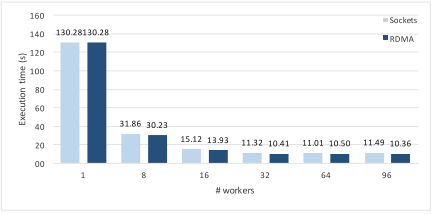
(a) Query 1 on 4 DQE instances.



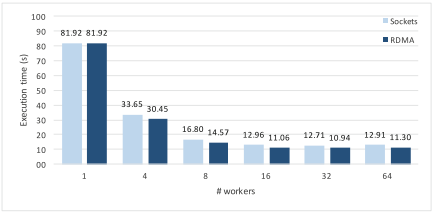
(b) Query 1 on 8 DQE instances.



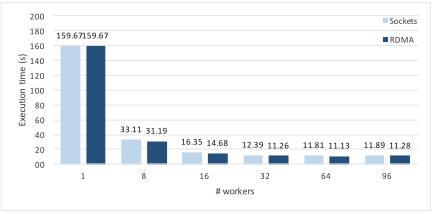
(c) Query 2 on 4 DQE instances.



(d) Query 2 on 8 DQE instances.



(e) Query 3 on 4 DQE instances.



(f) Query 3 on 8 DQE instances.

Fig. 5. Execution times of the analytical queries for the sockets and RDMA middleware, when varying the number of DQE instances and the number of workers.

5 Related Work

Different works explored previously the use of RDMA technologies (and the RDMA Verbs interface) to improve performance of communications in parallel/distributed systems. Liu *et al.* [9] proposed an implementation for MPI [4] (the *de facto* standard for communication in the high-performance computing field) based on RDMA writes. Similar to our approach, they use a pool of pre-allocated buffers, and there is also a match between buffers on sender and receiver sides, so that the state of the receiver buffer can be "predicted" on the sender side. Whereas we use a single pair of matching circular buffers, they use a pool of buffers organized as a ring. Moreover, they rely on active polling to detect incoming data, which limits scalability. Therefore, they limit RDMA communication to sub-sets of processes, and use send/receive requests for the remaining communications. Sur *et al.* [14] proposed a different approach. They use a *rendezvous protocol*, where the transfer of data is first negotiated through send/receive requests, and then RDMA read requests are used to transfer data. This

approach is not appropriate to our case, where we have multiple threads communicating concurrently through the same connection. We choose to use matching buffers on sender and receiver, together with notifications of read (processed) data on the receiver, to simplify the coordination between sender and receiver side, which is particularly important to reduce contention when multiple threads may try to send data concurrently (an issue not discussed by [14]).

RDMA was used to implement FaRM [3], a distributed computing platform which exposes the memory of multiple machines of a cluster as a shared address space. FaRM design has similarities with the solution we propose: It also relies on circular buffers and RDMA write requests to transfer data, for example. However, it uses active polling to detect received data, and RDMA writes to notify the sender of data removed from the buffer by the receiver side. Moreover, FaRM is designed for minimal latency, whereas in our case we take advantage of the asynchronous nature of the application using the middleware to batch multiple messages, increasing latency, but reducing communication overheads, and improving the overall application performance.

In the recent years, RDMA was also explored to improve different components of well-known software stacks for distributed computing. Wang *et al.* [18,19] provide an alternative shuffling protocol for Hadoop implemented using RDMA, which uses send/receive requests to request data, and then RDMA write requests to transfer the data. Lu *et al.* [10] also used RDMA to improve the performance of the Hadoop RPC communication.

6 Conclusions

In this paper we presented an RDMA-based communication middleware to support asynchronous shuffling in parallel execution of analytical queries, discussing the alternatives considered and the insights acquired with its implementation. When compared with a previous sockets-based middleware implementation, experimental results show that our new RDMA implementation enables a reduction of communication costs of 75 % on a synthetic benchmark, and a reduction of as much as 14 % on the total execution time of analytical queries, even when using a software implementation of the RDMA protocol, showing that redesigning communications to follow an RDMA approach can provide considerable benefits.

Acknowledgements. This research has been partially funded by the European Commission under projects CoherentPaaS and LeanBigData (grants FP7-611068, FP7-619606), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC-2894), the Spanish Research Agency MICIN project BigDataPaaS (grant TIN2013-46883), and the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project POCI-01-0145-FEDER-006961.

References

1. Darema, F.: The SPMD model: past, present and future. In: Cotronis, Y., Dongarra, J. (eds.) PVM/MPI 2001. LNCS, vol. 2131, p. 1. Springer, Heidelberg (2001)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
3. Dragojević, A., Narayanan, D., Castro, M., Hodson, O.: FaRM: fast remote memory. In: USENIX Symposium on Networked Systems Design and Implementation, pp. 401–414 (2014)
4. Forum, M.P.I.: MPI: A message-passing interface standard. University of Tennessee, Technical report (1994)
5. Gonçalves, R.C., Pereira, J., Jimenez-Peris, R.: Design of an RDMA communication middleware for asynchronous shuffling in analytical processing. In: CLOSER - CoherentPaaS/LeanBigData Projects Workshop (to appear)
6. Apache Impala project. http://impala.io
7. Jimenez-Peris, R., Patino-Martinez, M., Kemme, B., Brondino, I., Pereira, J., Vilaça, R., Cruz, F., Oliveira, R., Ahmad, Y.: CumuloNimbo: a cloud scalable multi-tier SQL database. Data Eng. **38**(1), 73–83 (2015)
8. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. **32**(4), 422–469 (2000)
9. Liu, J., Wu, J., Panda, D.K.: High performance RDMA-based MPI implementation over InfiniBand. Int. J. Parallel Program. **32**(3), 167–198 (2004)
10. Lu, X., Islam, N.S., Wasi-Ur-Rahman, M., Jose, J., Subramoni, H., Wang, H., Panda, D.K.: High-performance design of Hadoop RPC with RDMA over InfiniBand. In: International Conference on Parallel Processing, pp. 641–650 (2013)
11. MacArthur, P., Russell, R.D.: A performance study to guide RDMA programming decisions. In: ACM International Conference on High Performance Computing and Communication & IEEE International Conference on Embedded Software and Systems, pp. 778–785 (2012)
12. Mellanox Technologies: RDMA Aware Networks Programming User Manual (2015)
13. Stuedi, P., Metzler, B., Trivedi, A.: jVerbs: ultra-low latency for data center applications. In: 4th Annual Symposium on Cloud Computing, pp. 10:1–10:14 (2013)
14. Sur, S., Jin, H.W., Chai, L., Panda, D.K.: RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 32–39 (2006)
15. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow. **2**(2), 1626–1629 (2009)
16. Transaction Processing Performance Council: TPC Benchmark C Standard Specification, Revision 5.11 (2010)
17. Trivedi, A., Metzler, B., Stuedi, P.: A case for RDMA in clouds: turning supercomputer networking into commodity. In: Asia-Pacific Workshop on Systems (2011)
18. Wang, Y., Que, X., Yu, W., Goldenberg, D., Sehgal, D.: Hadoop acceleration through network levitated merge. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 57:1–57:10 (2011)
19. Wang, Y., Xu, C., Li, X., Yu, W.: JVM-bypass for efficient Hadoop shuffling. In: International Symposium on Parallel and Distributed Processing, pp. 569–578 (2013)