Design of an RDMA Communication Middleware for Asynchronous Shuffling in Analytical Processing

Rui C. Gonçalves[1], José Pereira[1] and Ricardo Jiménez-Peris[2]
[1]*HASLab, INESC TEC & U. Minho, Braga, Portugal*
[2]*LeanXcale, Madrid, Spain*

Keywords: Shuffling, Analytical Processing, Middleware, RDMA.

Abstract: A key component in a distributed parallel analytical processing engine is *shuffling*, the distribution of data to multiple nodes such that the computation can be done in parallel. In this paper we describe the initial design of a communication middleware to support asynchronous shuffling of data among multiple processes on a distributed memory environment. The proposed middleware relies on RDMA (Remote Direct Memory Access) operations to transfer data, and provides basic operations to send and queue data on remote machines, and to retrieve this queued data. Preliminary results show that the RDMA-based middleware can provide a 75% reduction on communication costs, when compared with a traditional sockets implementation.

1 INTRODUCTION

The proliferation of sensors networks or web platforms supporting user generated content, in conjunction with the decrease on the costs of storage equipments, lead to a significant increase of the rate of data generation.

This explosion of data brought new opportunities for business, which can leverage this data to improve its operation. On the other hand, storing and processing these massive amounts of data poses technological challenges, which lead to the emergence NoSQL database systems and solutions based on the MapReduce (Dean and Ghemawat, 2008) programming model, as an alternative to the traditional Relation Database Managements Systems (RDBMS) in large scale data processing.

An important concept in several frameworks for large scale data processing (e.g. Hadoop MapReduce (Hadoop,), FlumeJava (Chambers et al., 2010), Apache Storm (Storm,)) is data *shuffling*. Shuffling redistributes data among multiple processes, namely to group related data objects in the same process. Even though the basic concept is simple, different frameworks use different approaches to implement shuffling. For example, there are pull-based solutions, where the receiver process requests data from the source process, or push-based solutions, where the source pushes the data to the receiver. Multiple strategies may also be used to organize the data and to select the receiver process. For example, data objects may be distributed randomly or based on a hash function. The shuffling process may also sort the data objects of each process.

In this paper we propose a Java communication middleware designed to support efficient asynchronous data shuffling, using a push-based approach, which takes advantage of RDMA (Remote Direct Memory Access) for communication. It was designed to support hash shuffling on an analytical processing application, which was previously relying on Java sockets.

RDMA protocols provide efficient mechanisms to read/write data directly from the main memory of remote machines, without the involvement of the remote machine's CPU (at least when the RDMA protocol is directly supported by the network hardware). This enables data transfers with lower latency and higher throughput.

The proposed design relies on the RDMA Verbs programming interface, and uses one-sided write operations to transfer data, and send/receive operations to exchange control messages.

2 RDMA BACKGROUND

RDMA technologies (Mellanox, 2015) provide reliable data transfers with low latency/high-throughput,

Gonçalves, R., Pereira, J. and Jimenez-Peris, R.
Design of an RDMA Communication Middleware for Asynchronous Shuffling in Analytical Processing.
In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) - Volume 1*, pages 348-351
ISBN: 978-989-758-182-3

by avoiding memory copies and complex network stacks. Moreover, as applications access the network adapter directly when they need to exchange data, there is no need for the operating system intervention, which also reduces CPU utilization.

The RDMA Verbs is the basic programming interface to use RDMA. It provides two types of communication semantics: memory semantics or channel semantics. The former relies on one-sided read and write operations to transfer data. The latter relies on typical two-sided send/receive operations, where one side of the communication executes a send operation, and the other side executes a receive operation. It should be noted that the receive operation must be initiated before the send operation.

Network operations in RDMA Verbs are asynchronous. Requests for network operations (e.g., write, send) are posted on *queue pairs* (each one comprised of a send and a receive queue) maintained by the network adapter. A queue pair is associated with one connection between two communication end-points. The application may choose to receive *completion events* when requests are finished, which are posted into a *completion queue* associated with the queue pair. Moreover, the application may request to be notified when a completion event was added to the queue (these notifications are sent to a *completion channel*). This is useful to avoid the need of active polling the completion queues.

RDMA Verbs works with locked memory, i.e., the memory buffers must be registered. Besides locking the memory, the registration process also provides a security mechanisms to limit the operations that can be performed on each buffer. Network adapters have on-chip memory that can be used to cache address translation tables and other connection related data. Due to the limited amount of on-chip memory, the number of connections and the amount of registered memory used must be carefully decided (Dragojević et al., 2014).

The RDMA protocol and its Verbs programming interface may be supported directly by the network hardware, but it may also be provided by software (e.g., Soft-iWARP, Soft-RoCE). Even though these solutions do not provide some of the typical advantages of RDMA (e.g., they still require the involvement of the remote machine's CPU on network operations), they can provide improved performance as buffers are guaranteed to use locked memory, and by reducing system-calls.

In our prototype implementation, we are using the jVerbs library (Stuedi et al., 2013), 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 Java Virtual Machine to reduce memory copies, even when using an RDMA protocol implemented by software.

3 RDMA COMMUNICATION MIDDLEWARE

The goal for the communication middleware is to provide efficient data exchange between multiple threads, running on multiple processes. RDMA was previously explored by Wang et al. (Wang et al., 2013) for shuffling in Hadoop MapReduce. Their implementation followed a synchronous pull-based approach, and used send/receive requests to request data, and then RDMA write requests to transfer the data. In that case the data is produced in one phase, and consumed in a later phase, which means data to shuffle is likely to need to be stored on disk, to avoid blocking threads. Our proposal was designed for applications where data to shuffle is being produced and consumed at the same time (as it is the case of Apache Storm, for example). We also assume that data is produced and consumed at a similar rate (i.e. buffers rarely fill up), thus in our design data is never sent to disk. Instead, in case a buffer fills up, the thread using it will block.

On the base of our proposed design we have *shuffle queues*. They are used to asynchronously receive data objects from other processes (and its threads). That is, the shuffle queues abstract a set of queues used by a thread to receive data objects from the threads running on remote processes.

For threads running on the same process, data objects can be exchanged directly using shared memory and dynamic queues. However, when sending data objects to remote threads, the use of the network is required. In those cases, a thread maintains an incoming and an outgoing buffer per each remote thread. When sending data objects to a remote thread, they are initially serialized to the appropriate outgoing buffer (considering the target thread). The communication middleware provides the functionality of transferring data from the outgoing buffer of a thread to the matching incoming buffer of the target thread, from where the data objects will eventually be pulled by the remote thread.

In summary, the communication middleware was designed to provide the following functionalities:

- ability to send and queue data objects to remote threads;
- ability to pull queued remote data objects;

- ability to block a thread when there is no data objects to process (and to wake it up when new data objects become available); and
- ability to block a thread when local buffers are full (and to wake it up when space becomes available).

The RDMA middleware uses mainly one-sided RDMA write operations to transfer data objects directly between Java memory buffers. Additionally, it also uses send/receive operations to exchange control data.

When initializing the application, communication end-points are created on each process, i.e., an RDMA server connection is created, and bound to the machine IP. The next task is to connect the network. Briefly, this comprises the following steps:

- allocation and registration of memory buffers;
- allocation of queue pairs, completion channel, and completion queue;
- start of a new thread (the network thread), which handles the completion events;
- establish RDMA connections with all other processes;
- exchange of memory keys between processes (required to allow the execution of one-sided RDMA write operations), using send/receive operations.
- pre-allocation and initialization of objects needed to execute the network requests.

When shuffling data, threads send and receive data objects asynchronously using incoming and outgoing buffers to serialize data objects and to temporarily store them until they are transferred/pulled. These buffers are implemented as *circular buffers*. They have an head and a tail (new data is written at the head position, that is, the data available in the buffer is stored between the tail and the head).

When sending data objects to remote threads, the object is serialized to the outgoing buffer, and an RDMA write request is posted (queued for execution), to transfer a segment of data to the appropriate remote incoming buffer. The thread only queues the RDMA write request, i.e., it does not have to wait for the request to be actually executed. As there is no intervention of the receiving side, send/receive requests are used to notify the remote process that a data object was written in its buffers. This is done by the network thread, after it receives an event confirming that the RDMA write requests completed successfully. Moreover, the network thread will also update the tail of the outgoing buffer from where the data was transferred, as the space occupied by the data sent can now be reused.

Before posting the RDMA write request, the thread needs to determine whether there is free space available on the remote buffer. This is determined by the tail position of the remote before, which is tracked on the sending side (notifications are also used to update this information). If there is no space available on the remote buffer, the thread continues its operation, and the network thread will post the RDMA write request when it receives a notification updating the tail of the remote buffer.

The local outgoing buffers may also become full. When this happens, the thread blocks, as it cannot serialize its current object and proceed to the next one. When the network thread is notified that an RDMA write request completed, and space was released on the desired buffer, the network thread wakes up the blocked thread.

The data objects transferred will eventually be pulled by the receiver thread. The threads do not know when data was transferred into their buffers. To overcome this limitation, network threads exchange notifications when data is transferred. The network thread maintains a queue of buffers with data available for each thread, which allows the threads to avoid the need to actively poll all incoming buffers. If a thread has no data objects to process, it blocks. It is the network thread that will wake up this thread when additional data arrives. That is, the shuffle queues act as blocking queues. This design enables the overlap of communication and computation, as long as data is produced at a similar rate as it is consumed.

A prototype implementation of the proposed middleware design was implemented, and compared with a previously used middleware based on sockets, to provide a preliminary evaluation of the benefits of using RDMA. The sockets middleware used a similar push-based approach using circular buffers, but relied on non-blocking Java sockets between each pair of buffers to transfer data from outgoing to incoming buffers. This preliminary implementation for the RDMA middleware provided a reduction of communication costs of around 75%, when shuffling data among 32 threads on 8 machines, and using a software implementation of the RDMA protocol (Soft-iWARP).

4 CONCLUDING REMARKS

In this paper we proposed the design of an RDMA-based communication middleware to support push-based asynchronous shuffling.

Preliminary results, based on a prototype implementation of the RDMA-based middleware, show that

the RDMA approach proposed can provide a reduction of communication costs of around 75%, when compared with a sockets implementation. This preliminary work shows that we can benefit significantly from RDMA technologies, and that this is a research direction worth exploring.

ACKNOWLEDGEMENTS

This work has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 619606, project LeanBigData – Ultra-Scalable and Ultra-Efficient Integrated and Visual Big Data Analytics (http://leanbigdata.eu).

REFERENCES

Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. (2010). Flumejava: Easy, efficient data-parallel pipelines. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–375.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. (2014). FaRM: Fast remote memory. In *NSDI '14: 11th USENIX Symposium on Networked Systems Design and Implementation*, pages 401–414.

Hadoop. Apache hadoop project. http://hadoop.apache.org.

Mellanox (2015). *RDMA Aware Networks Programming User Manual*. Mellanox Technologies.

Storm. Apache storm project. http://storm.apache.org.

Stuedi, P., Metzler, B., and Trivedi, A. (2013). jverbs: Ultra-low latency for data center applications. In *SOCC '13: Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 10:1–10:14.

Wang, Y., Xu, C., Li, X., and Yu, W. (2013). Jvm-bypass for efficient hadoop shuffling. In *IPDPS '13: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 569–578.