Deconstructing the Shuffle operation in Big Data Workloads

1. Introduction 1

2. Motivation 2

3. Analysis of Shuffle in Spark 4

4. Implementation 5

   Socket based TCP/IP 5

   RDMA 7

5. Evaluation 8

   Cost on time 8

   Cost on CPU Utilization 10

   Cost on deserialization 11

6. Conclusion 12

1. Introduction

Many important applications in diverse domains require analyzing huge datasets. These datasets may be a relational table, a stream of events, or graph-structured data. To enable this analysis, several frameworks (e.g. Spark, MapReduce, Flink, GraphX) have been developed. In most of these frameworks, a user writes a program that has an analysis logic for a partition of the data. The framework takes in this program, partitions the data to be analyzed across multiple servers, and executes the program in parallel on these partitions. A key detail is that analysis programs may consist of multiple such computations that can be parallelized. The dependency between these computations can be captured using a DAG model, where a node represents a computation task and an edge represents a dependency between computations. Thus, in the DAG model, a computation can be triggered or started once all of the upstream computations have been done.

Big data frameworks may schedule computation tasks for different nodes of the DAG on different servers. To proceed with its computation, a node must have results of computations from upstream nodes available with it. This requires moving intermediate data from multiple servers to the nodes where data is required, an operation which is termed as the *shuffling* of

data. Since shuffling involves moving potentially large amounts of data across the network, shuffling results in several overheads as listed below -

- Shuffling costs can dominate the overall running time of an analysis program.
- Shuffling burns a significant amount of CPU cycles. To shuffle, a host CPU must determine the target location of intermediate data for downstream tasks. Post this, the host CPU must interact with the networking stack to initiate the intermediate data transfer. On the receiver end, the CPU gets involved in receiving and buffering the intermediate data. Additionally, the CPU is involved in deserializing the data so as to make it available for consumption to the downstream task.

We view shuffle as a service that a network should offer rather than one that is tied to applications. In this project, we wish to explore techniques to reduce shuffling costs and reduce the CPU utilization of the shuffling operation and provide the abstraction of a Shuffling-as-a-service operation. We explore what pieces of the shuffle operation can benefit from hardware acceleration. To this end, we do the following -

- First, we analyze the overheads incurred by shuffling. We analyze some common applications on Apache Spark and measure the overheads of shuffling. We use the TPC-DS benchmark on Spark SQL, PageRank computation on GraphX, windowed word count analysis of a large incoming stream of data on Spark Streaming. Our analysis shows that shuffle costs are significant and improvements in shuffle performance can help reduce the end-to-end running time of big data computations.
- We then study the internal architecture of the shuffle on Apache Spark and implement standalone prototypes that emulate the shuffle operation using TCP/IP and RDMA as transport substrates.
- We run micro-benchmarks using these prototypes and obtain some key insights that can guide the design of new shuffle-as-a-service network operation. We offer insights as to what components of the shuffle can benefit from hardware offload.

2. Motivation

To measure the impact of shuffle on big data workloads, we benchmark some common applications atop Spark as described below -

- **TPC-DS using Spark SQL** - TPC-DS is a popular benchmark used for measuring the performance of systems that serve OLAP or decision support workloads. We set up TPC-DS with a scale factor of 1, which means that the database has approximately 1GB of data. We then evaluate the shuffle performance on queries 12, 21, 50, 71, 85. These represent different classes of queries with different access patterns.
- **Word count using Spark Streaming** - We benchmarked the shuffle cost for streaming workloads by using a simple word count program that prints the word count every 2 seconds. We replay a large text file from HDFS as the workload and feed it to Spark Streaming using Netcat. We run the streaming workload for 5 minutes.

- **PageRank using GraphX** - We benchmarked the shuffle cost on PageRank atop GraphX using the Berkeley-Stanford dataset. The graph has 685230 nodes and 7600595 edges. We ran PageRank for 40 iterations.

Spark UI exposes a bunch of metrics that give insights into the shuffle performance. We use these metrics to perform our analysis. Spark gives us the below metrics with respect to shuffling -
(i) **Read blocked time (ms) -** This is the amount of time a downstream node in the computation graph is waiting for data to be read and shuffled from upstream nodes.
(ii) **Remote Reads (MB) -** This is the amount of data read from remote nodes during a shuffle.

We run the 3 workloads mentioned above which represent batch processing, stream processing, and graph processing. We dump metrics into a folder, persist it using Spark provided options, and analyze the metrics using the Spark history server. The results are presented below -

Workload	Time spent in shuffle (ms)	Remote data shuffled (MB)	Total running time (ms)	% time spent in the shuffle
TPC-DS Query 12	124 ms	111 MB	1243 ms	9.97%
TPC-DS Query 21	133 ms	198 MB	1781 ms	7.47%
TPC-DS Query 50	141 ms	177 MB	3914 ms	3.6%
TPC-DS Query 71	100 ms	91 MB	488 ms	20.49%
TPC-DS Query 85	112 ms	112 MB	1886 ms	5.94%
Word count	20ms	1.2MB	69ms	29%
PageRank	116 ms	195 MB	40 secs	0.7%

We observe that nodes in the computation stalls waiting for data to be shuffled. The percentage of time is particularly high in the case of streaming workloads, where low latency is of paramount importance. Thus, reducing shuffle costs is an interesting problem that needs to be solved. In order to do this, we analyze the current architecture of shuffle in Apache Spark.

3. Analysis of Shuffle in Spark

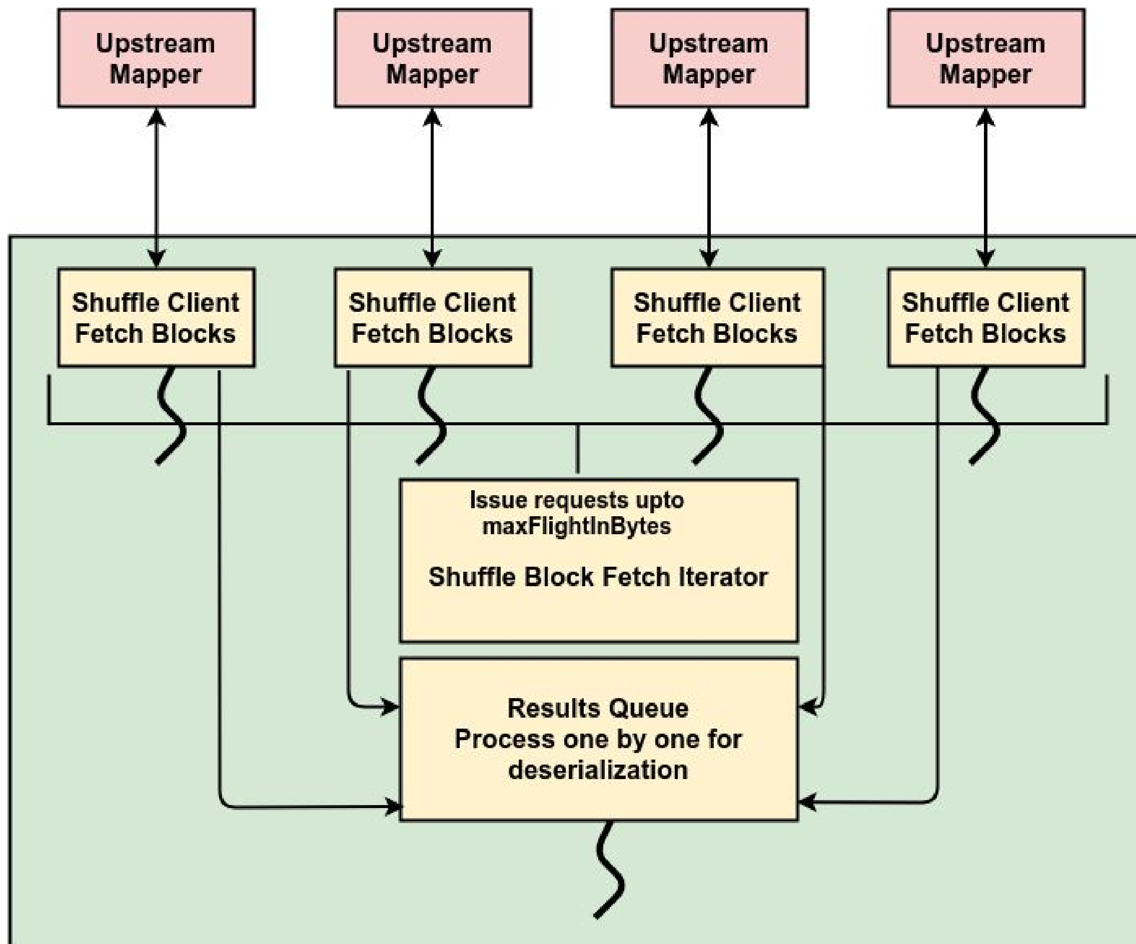

Figure 1: Anatomy of a shuffle operation in Apache Spark

Figure 1 shows the internal operation of a shuffle in Apache Spark. We obtained this analysis both by going through the Spark code as well as through multiple blog posts.

The key data structure used in the shuffle is the results queue, which buffers serialized data which is fetched by multiple threads over the network. A single thread consumes items from the results queue, deserializes the data, and computes some function over it. Generically, this function may be thought of as constructing a hashmap of key-value pairs that needs to be handed over to the reducer logic.

The shuffle operation begins with Shuffle Block Fetch Iterator issuing remote read requests to the upstream map tasks from which data needs to be fetched. To facilitate this, the Shuffle Block Fetch Iterator interacts with a Shuffle Client Fetch Blocks module. Each such module is responsible for issuing network requests to a single upstream mapper and enqueuing the

responses into the results queue. Each Shuffle Client Fetch Block runs on its own dedicated CPU thread. The Shuffle Block Fetch Iterator limits the total number of bytes of data that are outstanding on the network and waiting for deserialization on the results queue to a "maxFlightInBytes". This is to ensure that the executor does not run out of memory due to the results queue having too much data enqueued.

An important design decision is on what the size of each individual request to an upstream mapper should be. Apache Spark limits each request to be a maximum size of maxFlightInBytes/5. This magic number of 5 was chosen so that Spark can concurrently fetch data from at least five machines at the same time and not incur heavy throughput penalties.

4. Implementation

Based on the design of shuffle as mentioned in section 3, we implemented the same using TCP/IP and RDMA to compare and contrast the tradeoff of offloading shuffle operations onto the network.

Socket based TCP/IP

Figure 2 shows the details of the Mapper and the Reducer process. The entire shuffle processing involves a set of Mapper (Server) processes running on nodes in a cluster with data in memory and a Reducer (Client) process in another node that intends to pull data from the mapper. So, when the mapper process starts, it instantiates a dictionary of key-value pairs and fills it with randomized data of specific size (configurable). Then, it starts listening for connections on the specified port.

The Reducer process, when it starts, reads a configuration file with details of servers (mapper) and spawn threads for each mapper (Communication threads) and pins them to individual cores. It also spawns a separate thread on a specific core (Grouper thread) that runs the deserialization and grouping (a function) logic over the received data. Each communication thread is assigned a remote mapper and it is responsible for all communication with the respective server. The mapper threads initiate TCP socket connection and send a request message intending to start the shuffle process. On receiving the request, the mapper spawns two threads – one to receive requests from client and another to send replies back to client with serialized key-value pair data. Then, it replies back Acknowledgement to the reducer confirming its readiness to start data transfer.

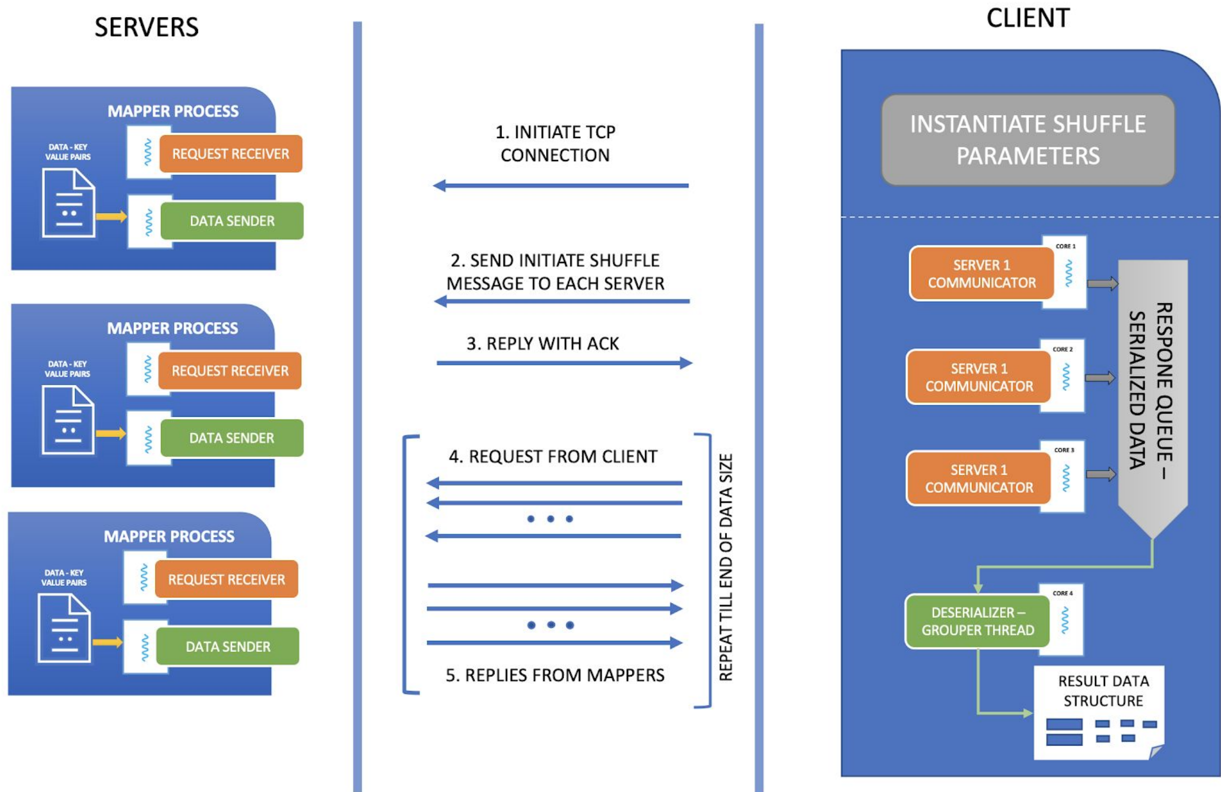

Figure 2: Description of TCP/IP implementation of Shuffle process and different steps involved in the communication

The reducer, then starts the data transfer by sending requests and gets back replies. The threads, on receiving serialized data, enqueues them on a shared queue. The grouper thread dequeues data from the queue, deserializes the received data, converts them into key-value pairs and inserts them into the result data structure. The result data structure holds the keys with a list of values received from the mapper. The reducer process limits the maximum requests sent to each server (as with the actual design of the shuffle in spark). The size of key, value pairs and number of records to reply determines the read length and maximum shuffle size in our experiments. We have implemented the design in C and used the Protocol Buffer library to serialize and deserialize the data exchanged between the server and the client.

RDMA

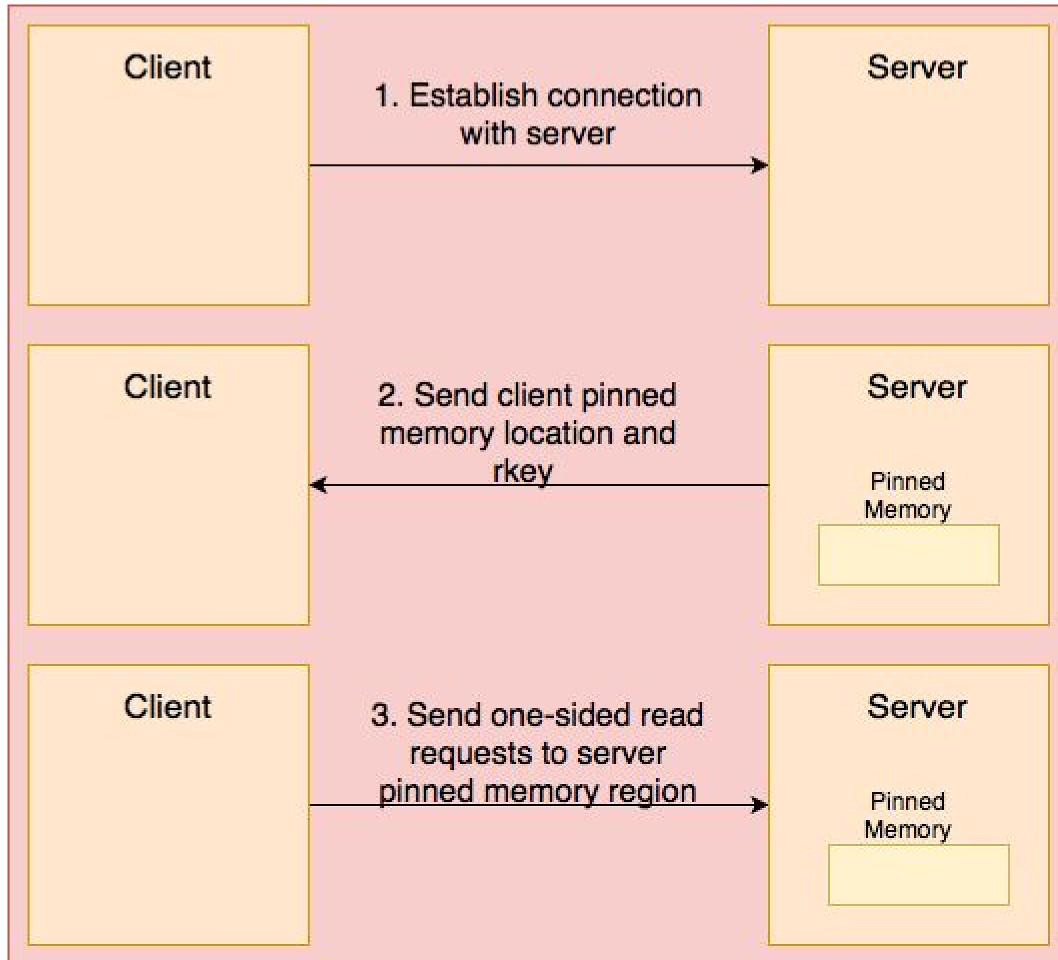

Figure 3: Shows the different steps involved in client (reducer) pulling data from the server (mapper) using RDMA operations

Client (reducer) establishes a connection with the server. Following that, the server (mapper) allocates memory which is pinned to the main memory and sends the client a pointer to the pinned memory region as well as a remote key, needed to read data from this memory region.
Once the client (reducer) has details about the memory region within the servers to pull data from, the client sends out one-sided read requests to the servers. The server CPU cycles are saved in the process, because the client just pulls data from the memory region, using one sided read operations. We also achieve kernel bypass on the client in the process.
The client has a granularity of read associated (read length) with each remote read operation, and reads the contents of the entire memory region over multiple remote read requests.

5. Evaluation

We ran our experiments in a cluster with 4 nodes in which 3 nodes assume the role of mapper that contains data and another node runs the reducer that pulls data from the mapper. To determine the impact of shuffle with respect to CPU processing and the time spent in communication between nodes, we varied the total size of shuffle data and the read length in each request. We measured the CPU utilization, time taken to complete each request and clock cycles spent during each operation. To ensure uniformity, all our experiments are run after setting the CPU frequency to performance mode. All experiments were repeated for both our implementation - TCP/IP and RDMA and the results were compared.

Cost on time

To measure the impact of shuffle with respect to time, we ran a set of 3 experiments. In the 1st experiment, we set the total data size to be 0.5 MB on each mapper and measured the total time taken to complete the shuffle.We set the maximum request in flight at a time to be 3. We repeated the experiment for data size of 1 MB and 5 MB. Figure 4, 5, 6 shows the comparison between TCP/IP and RDMA for the respective data sizes.

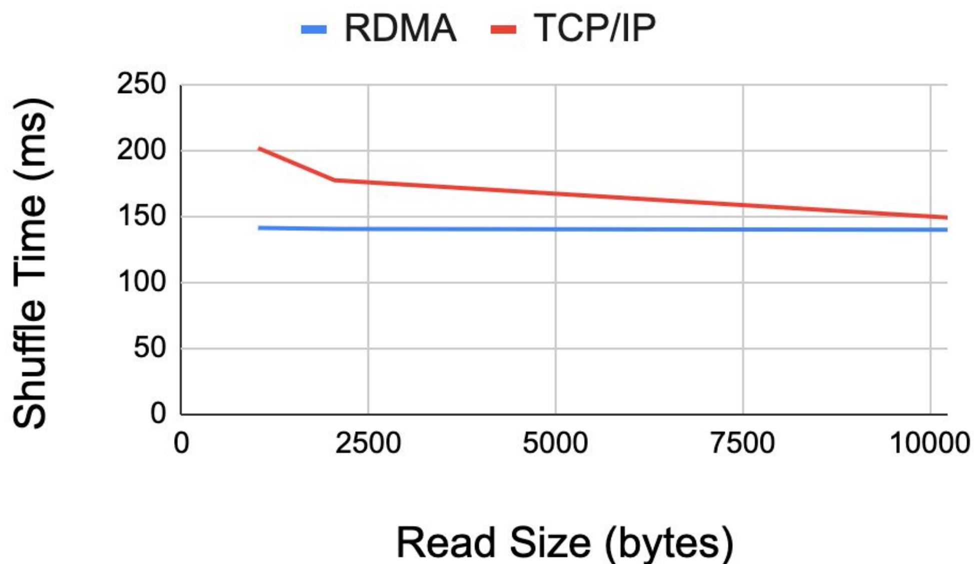

Figure 4: Variation of total shuffle time with different read size and total data size of 0.5 MB

Figure 4 shows the variation of total shuffle time with respect to different read size. The total size of the data retrieved is 0.5 MB. We observed that, in TCP implementation as the size of each reply increases, the total shuffle time reduces. This is because, with larger read size the total number of request-response decreases, and thus the time spent by the kernel processing the request and response packets reduces. With RDMA implementation, since RDMA operations are one-sided operations without the kernel interventions on the client, the time taken to complete shuffle stays almost the same despite the different read size.

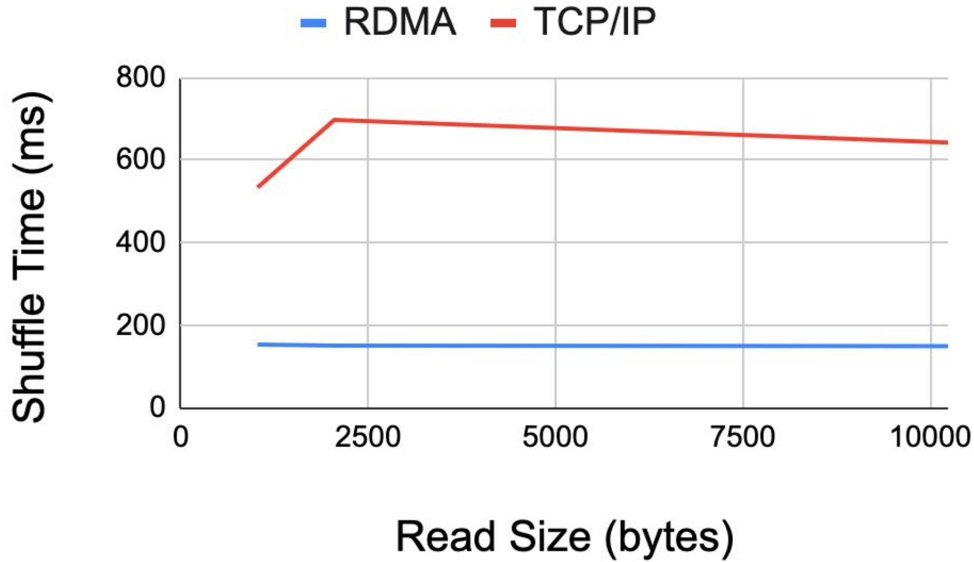

Figure 5: Variation of total shuffle time with different read size and total data size of 1 MB

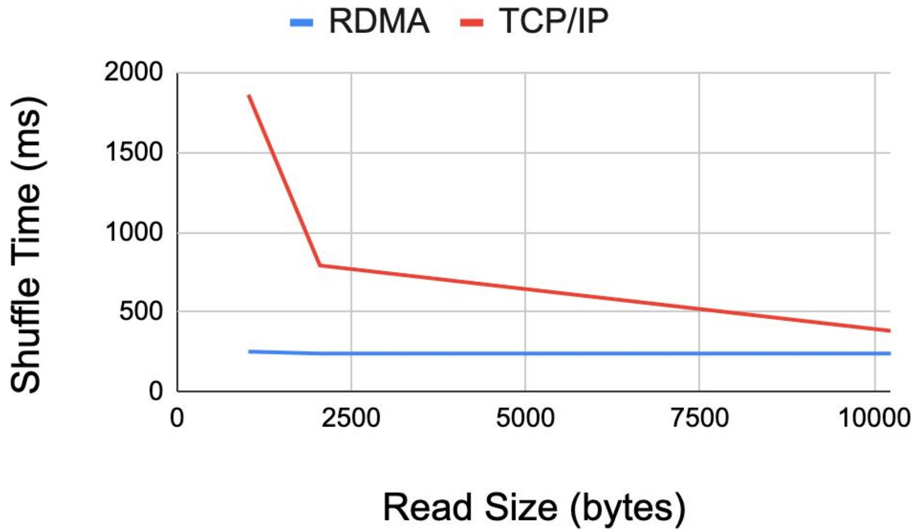

Figure 6: Variation of total shuffle time with different read size and total data size of 1 MB

Figure 5 shows the result for the experiment repeated for 1 MB of total data. With TCP, we find that when the read size is 2KB, the total completion time of shuffle increases. Further, investigation of metrics revealed that the time taken for total shuffle time is dominated by the deserialization thread. Same as in the previous experiment, RDMA has similar completion time across different read sizes and better total completion time irrespective of the read size. Figure 6 shows the result of the same experiment repeated for 5 MB of total shuffle data. Similar to experiment with 0.5 MB of data, we find that as the read size increases, due to the decrease in

number of requests, the total completion time of TCP decreases as a result of lower kernel overheads. With increase in total data size, RDMA has total completion time order of magnitude lesser than TCP for smaller read sizes.

Cost on CPU Utilization

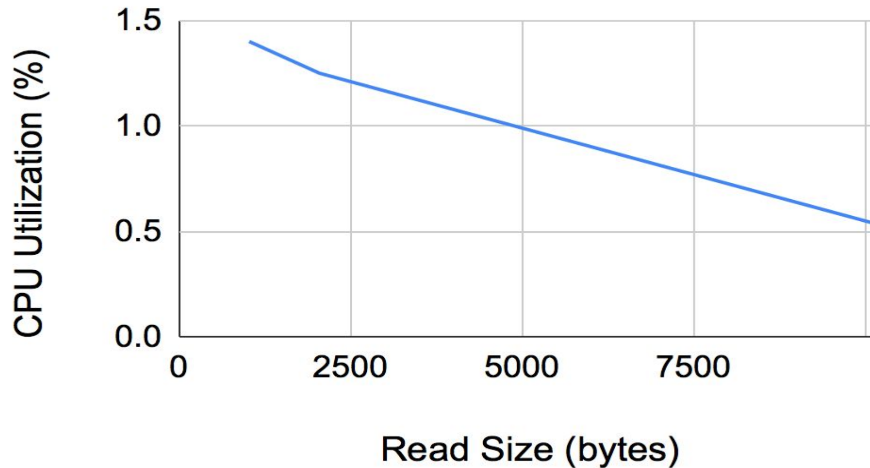

Figure 7: Variation of CPU utilization on the server with read length for TCP/IP implementation

Figure 7 measures the variation of CPU utilization with read length for a data size of 5MB for the TCP/IP implementation. For the TCP/IP implementation, we see that the CPU utilization on the server drops with increasing read size, because with increasing read size, the number of requests goes down. For the RDMA implementation, we observe the server utilization to be zero, because there is no CPU involvement as we pull data using one-sided reads.

Cost on deserialization

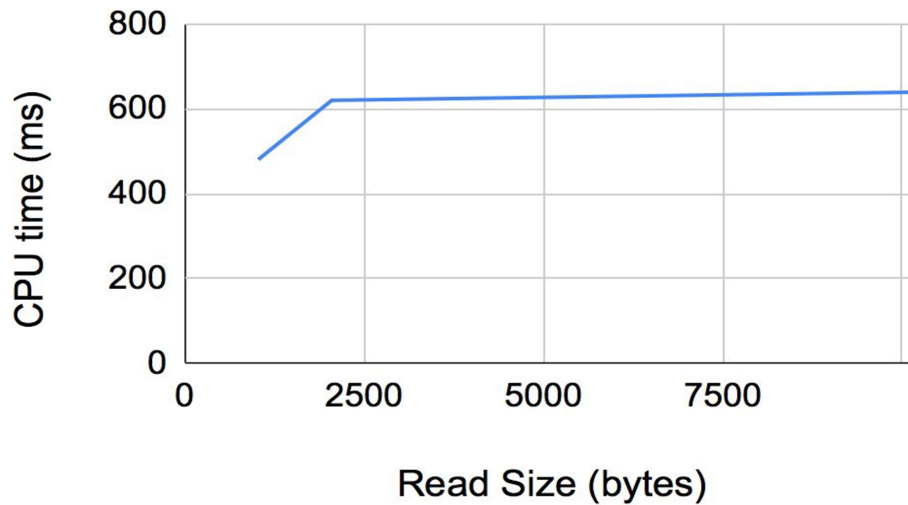

Figure 8: Variation of shuffle deserialization with read size for a TCP/IP implementation

Figure 8 measures the variation of deserialization time with read length for a data size of 1MB for the TCP/IP implementation. We see that deserialization time is not as sensitive to read length. It initially increases and saturates to a particular value.

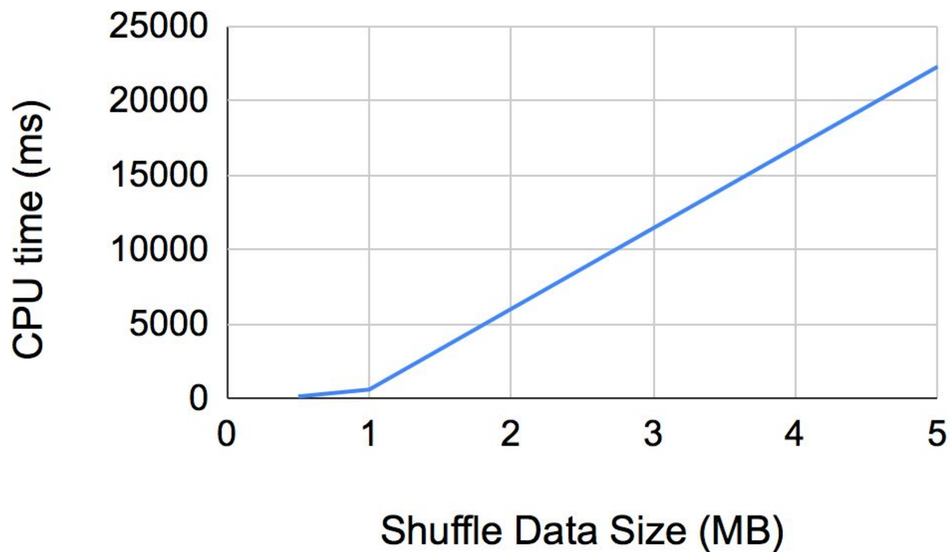

Figure 9: Variation of shuffle deserialization time with the size of data read

Figure 9 measures the impact of variation in the shuffle time with an increase in data size, for a fixed read length of 2048 bytes. As the data size increases, the CPU deserialization time also

increases. This is because the overhead associated with the deserialization of more data only goes up as we have more data that is being pulled from other machines.

6. Conclusion

RDMA helps accelerate the shuffle operation on the reducer primarily via kernel bypass. This is especially beneficial when the number of read requests necessary to collect the data from mappers is large, which would incur significant kernel overhead. Additionally, it also improves the CPU utilization on the mappers by freeing up CPU resources for other activity, because of the one-sided nature of the read operations that the reducer uses to accumulate the data from the mappers. On the contrary, usage of RDMA has the disadvantage of having to keep some memory on the mapper side pinned while the remote reads are in progress, which reduces the memory available to other processes in the system.

All our experiments show that the deserialization of data has a significant cost on CPU cycles. As the number of read requests increases, we see that the total completion time of shuffle is dominated by the thread which does deserialization of data and applying a user-desired function over it. Given such significant overhead of deserialization on the reducer side, we suggest that hardware offload of the deserialization could free up CPU cycles on the reducer side.