Influence of Parallelism Property of Streaming Engines on Their Performance

Nigel Franciscus[1], Zoran Milosevic[2] and Bela Stantic[1]

[1] Institute for Integrated and Intelligent Systems, Griffith University, Australia
[2] Deontik, Brisbane, Australia

Abstract. Recent developments in Big Data are increasingly focusing on supporting computations in higher data velocity environments, including processing of continuous data streams in support of the discovery of valuable insights in real-time. In this work we investigate performance of streaming engines, specifically we address a problem of identifying optimal parameters that may affect the *throughput* (messages processed/second) and the *latency* (time to process a message). These parameters are also function of the parallelism property, i.e. a number of additional parallel tasks (threads) available to support parallel computation. In experimental evaluation we identify optimal cluster performance by balancing the degree of parallelism with number of nodes, which yield maximum throughput with minimum latency.

1 Introduction

In Big Data environment, there are two types of data processing engines basically dedicated for different purposes, namely batch and streaming engines. Batch processing is concerned with handling of massive volume of data while streaming is concerned with processing data of high velocity. Several platforms and tools have been developed to support big data environments, of which the most widely known is Hadoop[3] which utilizes MapReduce batch processing. The Hadoop is however not adequate for real time streaming requirements, as for example is needed for stock market pattern monitoring, ad-serving [5] or real-time personalised recommendation [4]. Total time T to process a message is

$$T = B + M$$

where B is the time to collect data in the Hadoop input buffer and M as the time to process the data through MapReduce. It is obvious that some applications require near real-time analysis and this type of analysis can be supported by streaming engines. Two most popular Streaming Processing Engines (SPEs) are Storm and Spark Streaming which are fault tolerant and guarantee message delivery.

[3] Apache Hadoop, http://hadoop.apache.org/
[4] Spotify Labs, https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm/

Due to simultaneous threads execution, it is expected that degree of parallelism will improve the throughput. However, it has been mentioned in literature that when the degree of parallelism reaches certain threshold there may be a latency penalty which will impact overall performance [7]. In order to identify the optimal configuration and performance of SPEs with varying degree of parallelism, we have carried out a series of experiments across the clusters in Storm and Spark and looked closely in throughput and latency performance parameters. In experimental evaluation, which is based on counting words extracted from sentences submitted to the SPEs in continuous stream, we varied different *degrees of parallelism* and *number of nodes* used. In addition, we investigated possible relation between network bandwidth and threshold that each engine can support. We have not considered the impact of the CPU usage and memory consumption.

2 Related Work

To the best of our knowledge, there is no existing work focusing on the Streaming Processing Engines regarding to the degree of parallelism. However, some of the benchmarking has been done in term of performance.

Work on Discretized Stream provided a performance analysis of Spark and Storm throughput [10, 9]. Bell - Labs explored Storm inner core to get Storm performance optimisation by looking at parameter configuration [7, 1]. The inner core that has been tested includes number of parallelism hint, network usage, CPU usage and horizontal scalability. Bell - Labs have found that the configuration is critical and they are planning to build an engine which can determine the best configuration for every job automatically. The key attribute to consider in the implementation of the engine is the horizontal scalability configuration.

Our work is built on this specific criterion of *parallelism degree* which exploits the benefit of distributed systems by increasing the number of parallel threads and distribute them reliably across the nodes. However, it has been shown that the effect of spout-bolt parallelization has reach level of threshold with 24 degree of parallelism. Our work proved that we can overcome the level of threshold by investigating network bandwidth. Note that threshold is also dependent on environment hardware.

Analysis presented in [8] describes the use of Storm within Twitter including Storm architecture and methods. An investigation of comparison between Apache Storm and Apache Flink (Formerly known as Nephele) was presented in[4, 3] in relation to latency constraint and throughput. Since latency is a non-trivial property in streaming context, this work has studied on how the system accommodates such latency restriction while producing maximum throughput. While study conducted in [6, 2] provided architectural comparison of Spark and Storm, it concluded that Storm performs well in sub-second *latency* and no data loss while Spark performs better where *throughput* and stateful processing are required.

3 Influence of Parallelism Property

We have investigated the dependency of throughput and latency measures in streaming engines considered from i) the degree of parallelism parameters and ii) number of worker nodes as variables. We performed the analysis on the individual engine basis which subsequently led us to establish some comparative results of one-at-the-time and micro-batch processing. The purpose of these experiments was to better understand the interplay of various parameters that are configurable in both engines, and various choices that can be adopted to support performance tuning

Our experiment was conducted on a cluster of 20 nodes interconnected via Giga Ethernet. Each node has quad core Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz with 4GB RAM and 500GB SATA HDDs. The nodes run Scientific Linux release 6.3 and are connected using standard 100Mb/s Ethernet and 1Gb/s Ethernet. For the experiment, datasets are handled by Apache Kafka, where messages are injected into Kafka to build up queue. Our method to collect the statistics of *throughput* and *latency* is during the peak time of the message being processed. Thus, it will allow engine to apply the latency constraint while trying to achieve maximum throughput.

For each collection we calculate the mean of the average throughput-latency within the interval of biggest output with average of 5 minutes after job submission. We use both Storm and Spark metrics to collect statistics for both throughput and latency. In this experiment, we use Apache Storm 0.9.3, Apache Spark 1.4.0 and Apache Kafka 0.8.2.1. Both Apache Storm and Spark are given 4 slots (executors) allocation for each nodes.

As mentioned before, we use word count job as the measurement benchmark for testing Storm and Spark Streaming capabilities. This job imposes high processing demand on CPU resource. Each experiment is tested within the same cluster and same configuration for *degree of parallelism* parameter to facilitate comparison. Both engines have the degree of parallelism parameter that defines replication of receiver and processor to handle multiple streaming connections.

In order to get message reliability, we implement Apache Kafka as the distributor. Messages are generated in Kafka cluster before proceed to streaming engine. Each message consists of 30 - 35 words that has record size for about 5 - 10 bytes. It is important to note that degree of parallelism of Storm spout and Spark receiver is dependent on the number of topic partitions from Kafka brokers. In order to get desired degree of parallelism, each topic with different number of partition needs to be set upon implementation.

Our experiments with storm demonstrated significant contribution of the spout and bolt parallelization on single node. Figure 1 depicts the combination of spout and bolt parallelization resulting in linearly increasing throughput (tuple/sec). It can be seen that spout replication allows more tuple in the stream supporting increase in traffic and throughput. However, each replication requires more workers as spout replications use most of threads in single node (with 4 executors allocation) because spout component is the first in the topology and thus is first to be served through available slots. This has been noticed when degree of

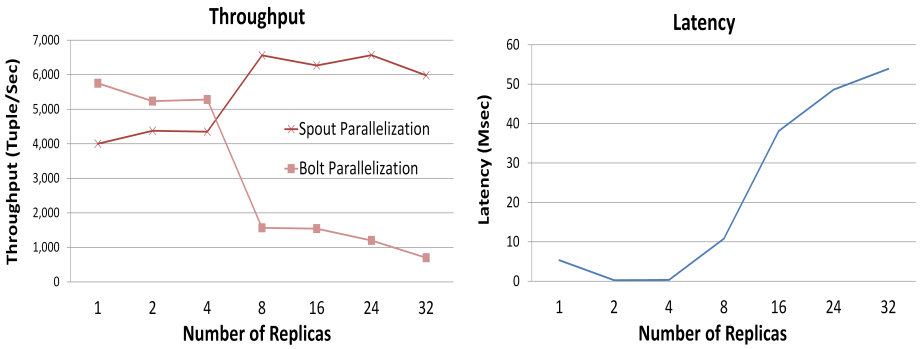

Fig. 1. Measuring Storm throughput and latency on single node.

parallelism reached 8. Since there are only 4 executors initially, bolt performance decrease dramatically with increased spout replication beyond threads threshold with 8 replicas. Figure 1 also indicates the latency for bolt degree of parallelism processing within single node. It was initially expected that latency will be stable throughout the duration of topology processing. However, we found that, due to overloading threads consumption, latency spikes drastically when bolt gets overloaded. This is because numbers of messages emitted from spout are building up the queue in internal buffer system that bolts access and bolt can only process as much as it is available in the buffer. Note that, it is also due to the fact that spouts have used all the available executors which results in 'overstressed' bolts. This suggests that Storm has a good respond in maintaining the topology under low level of resources. It is evident that there must be a restriction mechanism system similar to Spark to handle such circumstances.

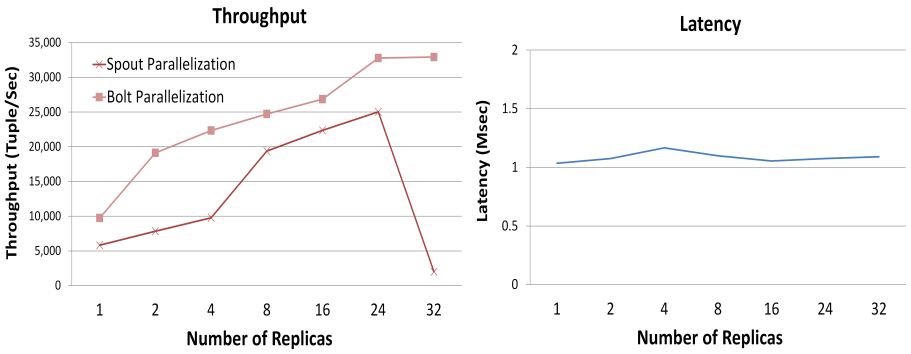

Fig. 2. Measuring Storm throughput and latency based on multiple nodes.

In order to compare one-at-the-time ability for horizontal scaling, we carried out a performance test with Storm on multiple nodes as a comparison with single node. Figure 2 indicates the horizontal scalability that supports data distribution and load balance through adding more worker nodes. With the increasing of degree of parallelism degree, more executors are needed to maintain number of threads spawned. Thus, there is need to try to balance the number of worker nodes with degree of parallelism (e.g. 4 degree of parallelism = 4 worker nodes). As the number of worker nodes increases to match degree of parallelism, throughput is steadily increased because for each increasing input stream in spout it will allow more tuples to be consumed by the topology.

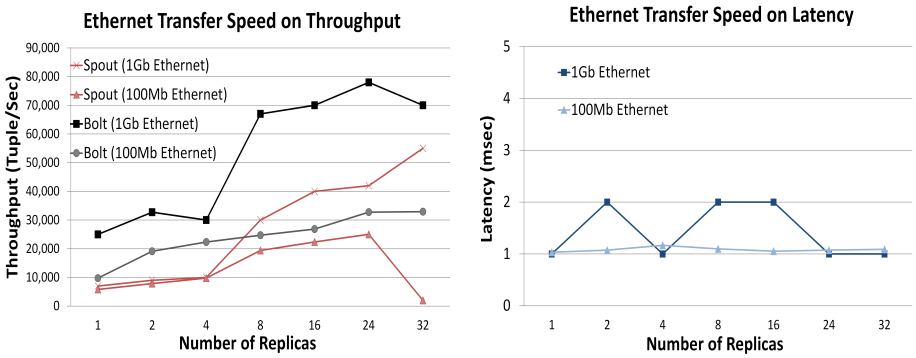

Fig. 3. Measuring Storm Throughput and Latency based on Ethernet switch transfer speed (100Mb and 1Gb).

In relation to latency, by adding more workers it has been proven to reduce the load of internal processing by distributing it across the cluster. This is achieved by sharing incoming tuple to each given workers to reduce bottleneck in the internal buffer. We have identified that by adding more worker nodes decrease overloading of bolts by exploiting more executors, which was also evident from the decrease in bolt capacity. Bolt capacity is measured 0 to 1 - where the capacity over 1 indicates overloading bolt and one should increase the degree of parallelism. Note that in part this is also due to data distribution across nodes which help Storm reducing excessive tuple queues in the internal buffer. Furthermore, it is worth to note that Storm internal processing (bolt) is powerful to handle huge amount of streaming data and then process it with very low latency less than 1 ms.

In terms of the impact of network protocol, since messages are delivered from socket to socket through TCP/UDP connection protocol, we expected that improvement in transport layer will affect the message deliveries. This was indeed the case. Figure 2 demonstrates that with 100Mb Ethernet switch, our cluster has reached the threshold when the *degree of parallelism* was 24, but with faster

network (1Gb) threshold was 32. Figure 3 indicates not only improvement in level of threshold, but also in overall performance for both throughput and latency.

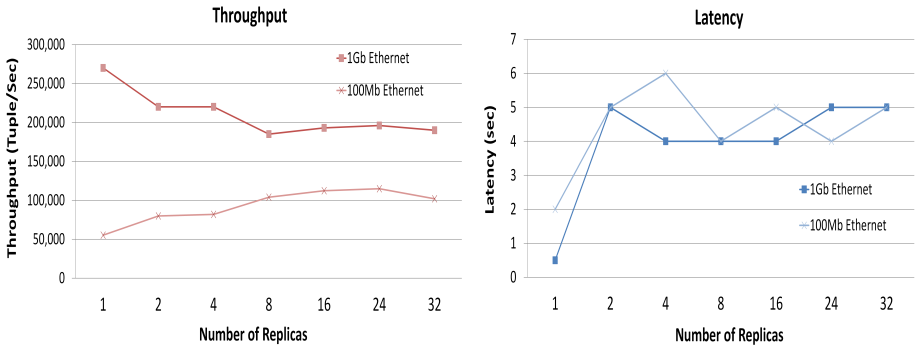

Fig. 4. Measuring Spark throughput and latency based on multiple nodes.

Our experiments have showed a significant *throughput* improvement from micro-batch compared to Storm. It may be noted that our prior experience has suggested insignificant differences in using different micro-batch interval size; the results are sufficiently general with the case of 1 second batch interval. Result shows that *degree of parallelism* does not significantly benefit neither *throughput* nor *latency*. Spark Streaming performance on multiple nodes is worse compared to the single node performance with default degree of parallelism configuration. In comparison to Storm, on single node Spark is able to perform 15 times better in terms of throughput and on average 4 times faster on multiple nodes. However, the trade-off from having big throughput is the increasing of latency into seconds. Note that latency on the order of seconds is not sufficient for process real-time analysis. Our experiment also shows that Spark Streaming will randomly choose worker nodes for the scheduler task. A drawback from this approach is that each opening and closing connection between master - workers will increase network traffic which affects the latency.

Figure 5 highlights the comparison between Storm and Spark based on maximum throughput and latency. Note that on Storm we measure the throughput based on spout as it represents how many tuples have been processed inside the topology. We performed several tests on different Spark micro-batching interval from 2 to 0.1 second. It is evident that, the micro-batch interval does not affect the maximum throughput (calculated in message per second) or latency. Spark Streaming does not benefit from degree of parallelism on multiple worker nodes.

Additionally, Spark has the restriction mechanism that will not allow for total number of executors to be less than the degree of parallelism, for example on a single node with 4 cores you cannot have more than 4 degrees of parallelism. On the other hand, in Storm, it would be possible to have 8 degrees of parallelism on the 4 core node, but this will put the workers into overloading state. Since Spark

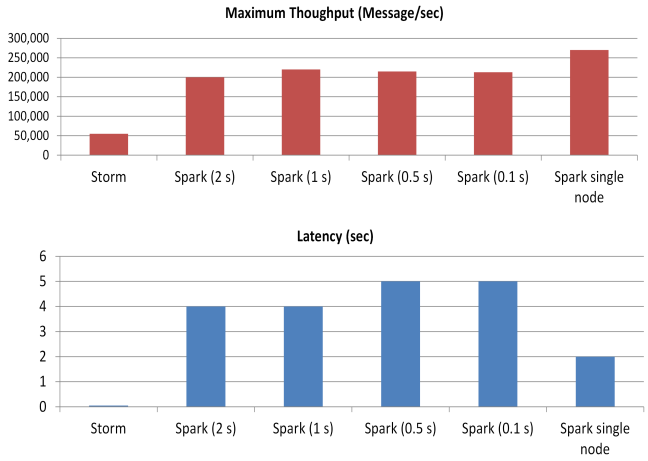

Fig. 5. The comparison between Storm and Spark in Throughput and Latency.

core framework exploits main memory its mini-batch processing can appear as fast as one at a time processing adopted in Storm, in spite the fact that the RDD units are larger than Storm tuples.

The benefit from mini-batch is to enhance the throughput in internal engine by reducing data shipping overhead such as lower overhead for ISO/OSI transport layer header which will allow the threads to concentrate on computation [3]. However, one may argue that the consequence of batching concept will add additional latency compared to one at a time. Since the processing of message transfer between worker nodes is performed in network transport layer, message transfer in batch will decrease network I/O. Thus, this will directly reduce CPU consumption and network bandwidth.

4 Conclusion and Future Work

In this work we evaluated performance of streaming engines as a variable of their degree of parallelism and number of worker nodes. We showed that by choosing appropriate degree of parallelism in combination with the right number of worker nodes one can achieve better performance. For example in case of 20 cluster nodes with four executors on each nodes optimal is 24 degree of parallelism, as after 24 the performance will drop.

We have also demonstrated that network bandwidth will increase performance of streaming engines and the level of threshold however also after the threshold the performance drops.

Another finding was that Storm has lower *latency* (in milliseconds) and lower *throughput* both on single and multiple nodes compared to Spark. Note that in Storm architecture, *throughput* can be improved by adding more worker nodes to allow more *degree of parallelism*. Spark micro-batch processing delivers high

throughput with high *latency* penalty (second). In our experiments, Spark had better *throughput* compared to Storm (15 times on single worker node and 4 times on multiple worker nodes with 4 executors on each node) but Storm delivered better latency over Spark.

In our future work, we plan to investigate the impact of configuring different number of *tasks*, which in these experiments was not explicitly defined, rather it was left to engine to select them. Second, we would like to explore the impact of *data locality* on performance. We would also like to consider the impact of the maximum tuples that a spout will allow to be processed in the topology on engine performance.

References

1. Bedini, I., Sakr, S., Theeten, B., Sala, A., Cogan, P.: Modeling performance of a parallel streaming engine: Bridging theory and costs. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering. pp. 173–184. ICPE '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2479871.2479895
2. Casale, G., Ustinova, T.: State of the art analysis (2015)
3. Lohrmann, B., Janacik, P., Kao, O.: Elastic stream processing with latency guarantees (2015)
4. Lohrmann, B., Warneke, D., Kao, O.: Nephele streaming: stream processing under qos constraints at scale. Cluster Computing 17(1), 61–78 (2014), http://dx.doi.org/10.1007/s10586-013-0281-8
5. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: Data Mining Workshops (ICDMW), 2010 IEEE International Conference on. pp. 170–177 (Dec 2010)
6. da Silva Morais, T.: Survey on frameworks for distributed computing: Hadoop, spark and storm (2015)
7. Theeten, B., Bedini, I., Cogan, P., Sala, A., Cucinotta, T.: Towards the optimization of a parallel streaming engine for telco applications. Bell Labs Technical Journal 18(4), 181–197 (2014)
8. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al.: Storm@ twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data. pp. 147–156. ACM (2014)
9. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. pp. 2–2. USENIX Association (2012)
10. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 423–438. ACM (2013)