Scale-up Graph Processing in the Cloud: Challenges and Solutions

Jasmina Malicevic
EPFL
jasmina.malicevic@epfl.ch

Amitabha Roy
EPFL
amitabha.roy@epfl.ch

Willy Zwaenepoel
EPFL
willy.zwaenepoel@epfl.ch

Abstract

Processing large graphs is an important part of the big-data problem. Recently a number of scale-up systems such as X-Stream, Graphchi and Turbograph have been proposed for processing large graphs using secondary storage on a single machine. The design and evaluation of these systems however have focused on physical machines. We expect that a natural evolution of such systems is to the cloud where a virtual machine would run the graph processing algorithm and access the graph from secondary storage remotely connected through the network. We evaluate a state of the art graph processing system called X-Stream in EC2 to identify challenges in this space. Our primary finding is that the network bandwidth between a virtual machine and remote storage becomes the limiter for performance. We show that this bottleneck can be somewhat alleviated through the use of VM local instance storage, network provisioning and compression.

Categories and Subject Descriptors D.1.3 [**Programming techniques**]: Concurrent Programming – *Parallel Programming*; D.4.2 [**Storage Management**]: Secondary Storage; E.1 [**Data**]: Data Structures – *Graphs and Networks*

General Terms Algorithms, Performance.

Keywords x-stream, large graph processing, cloud computing, storage, compression

1. Introduction

Graph processing has become an important analytics problem in the big data domain. Until recently processing large graphs had required large clusters or expensive supercomputers. This has changed with systems such as GraphChi and X-stream that enable the processing of large graphs on a single machine. This paper focuses on X-Stream, a state of the art graph processing system that can handle a Facebook social network sized graph on a single machine [1].

CloudDP'14, April 13 - 16 2014, Amsterdam, Netherlands
Copyright 2014 ACM 978-1-4503-2714-5/14/04…$15.00.
http://dx.doi.org/10.1145/2592784.2592789.

However, the amount of storage that can be attached to a single machine is limited. In this paper, we explore the possibility of running X-Stream in the cloud taking advantage of the large amount of storage that can be attached to a virtual machine, thereby breaking the barriers inherent in attaching storage to a single machine. Accessing storage in the cloud represents a fundamentally different environment from physically attached storage on a machine. The most important difference is the presence of an intervening network with its associated bandwidth and latency limitations.

In this paper we analyse the performance of X-stream on the Amazon EC2 [2] and Windows Azure [3] cloud environments and demonstrate the following:

- The network becomes the performance bottleneck when processing graphs from remote storage
- X-Stream's performance can be improved by
  - Using local instance storage for smaller graphs
  - Provisioning the network for better performance
  - Compression to reduce the amount of data transferred

The rest of this paper is structured as follows. In Section 2 we give a brief description of the X-Stream graph processing engine and its aspects that are of significance to this work. Section 3 describes the environment our experiments were set in as well as what our use case scenarios were. Section 4 covers our hypotheses and supporting experimental results. For generalization, we provide results from a subset of our experiments when run on Windows Azure [3]. The results are shown in section 5. Finally, we conclude in Section 7.

2. X-stream

X-stream is an edge-centric graph processing engine that sequentially iterates over edges. It provides a scatter-gather model allowing users to write vertex-centric programs consisting of a scatter and a gather step.

```
edge_scatter(edge e)
        generate update from e.source
gather(update u)
        apply update u to u.destination
while not done
        for all edges e
                edge_scatter(e)
        for all updates u
                update_gather(u)
```

Figure 1. Edge-centric Scatter-Gather

Figure 1 illustrates the programming model for X-Stream. In order to execute the scatter step, X-Stream iterates sequentially over the edge list and checks whether the source vertex has an update. If so it appends an update for the destination vertex to the update list. It then iterates over all updates and applies each update to the target vertex. This allows expressing a variety of graph algorithms. For example, in the case of the popular Pagerank algorithms, the update is a fraction of the current rank of the vertex, propagated via the edge to the targeted vertex. All the incoming updates are summed to compute the new rank for the vertex.

X-Stream partitions the graph such that the vertex data for each partition fits in RAM, while leaving edges and updates on secondary storage. Executing the programming model (Figure 1) for a graph partition then results in sequential access to secondary storage for edges and updates. As demonstrated in [1], the much larger sequential access bandwidth to secondary storage directly attached to a physical machine allows X-Stream with its unique programming model to outperform a number of graph processing solutions including Graphchi [4].

Our starting point for this paper is the implementation described in [1]. We added the necessary environmental support for X-Stream to start up and execute in the cloud. In addition, we implemented compression for the edge and updates lists. This was straightforward as we access both sequentially, this paper includes results from both the zlib [5] and snappy [6] compression algorithms.

3. Experimental Environment

We used the Amazon EC2 cloud infrastructure for our experiments; the main components of which are shown in Figure 2. Each physical machine (host computer) runs multiple virtual machines (instances). The instances share a physical NIC to connect to the network through which they can access remote storage (EBS volumes). Each physical machine also has a number of attached local disks (called instance store) that can be accessed by running instances. Instance storage is ephemeral lasting only for the lifetime of the VM, while EBS volumes are persistent across invocations of the VM. The EBS volumes can easily be detached and attached to another instance. In some cases the performance of the EBS volumes can be improved by striping the volumes into a RAID-0 array.

We used Amazon EC2 m1.large instances with 7.5GB of RAM and eight virtual CPUs, each approximately equal to a 1-1.2Ghz 2007 Opteron or Xeon processor. The VM runs a para-virtualized 64-bit Ubuntu Precise (server edition).

Amazon provides a cost-performance trade-off for accessing the EBS volumes as follows:

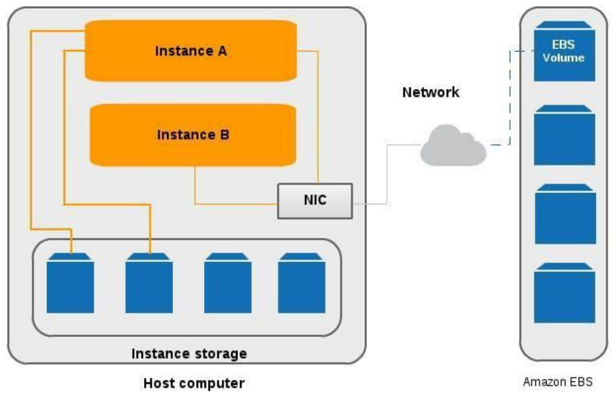

Figure 2. Amazon EC2 structure

- Since the network of the instance can be used for various purposes, IO requests can get delayed. The solution is to run an instance as "EBS optimized". This reserves 500Mb/s network throughput solely for communicating with the device.
- The disk holding the EBS volume is also shared between multiple tenants. EC2 therefore allows the user to pay for a "provisioned EBS volume". One can provision 30 IOPS per gigabyte. One IO operation cannot exceed 16KB and reaching peak provisioned IOPS requires the number of outstanding IOs to be maintained at least at 5 per 200 provisioned IOPS.

The combinations of storage type (instance and EBS), RAID and provisioning can be large. We pruned our experimental space down to the following types, eliminating those we knew would be outperformed by a different type on this list.

- Instance store (Instance)
- Two standard EBS volumes organized in a software RAID-0 array. No provisioning for either the network nor the IO device (EBS_S2S)
- Two provisioned EBS volumes organized in a software RAID-0 array accessed from a non EBS network optimized instance (EBS_S2P)
- Two EBS standard volumes in a RAID-0 array and an EBS optimized instance (EBS_P2S)
- Two EBS provisioned volumes in a RAID-0 array and an EBS optimized instance (EBS_P2P)

The number of provisioned IOPS was 1000 (500 for each volume).

4. Experiments

4.1.1 Characterizing the EC2 platform

We first characterized the EC2 platform from the perspective of X-Stream. We used the fio [7] tool to benchmark the bandwidth available to storage from the VM. In order the simulate the workload presented by X-Stream, we did sequential I/O by issuing a single synchronous request at a time varying the size of the request. This approximates X-Stream's disk access pattern that issues sequential I/O of constant configurable size to disk, using asynchronous I/O to ensure that there is always exactly one outstanding request to disk. The results are shown in Figures 3 for sequential reads and Figure 4 for sequential writes. We drew the following conclusions.

1. The fastest storage is the instance store. Moving to remote storage on EC2 therefore provides persistence and increased space in return for performance.

2. In terms of peak achievable bandwidth for sequential reads, provisioning the network has a bigger payoff than provisioning storage in the case of reads.

3. In terms of peak achievable bandwidth for sequential writes, provisioning the drives has a marginally better payoff than provisioning the network.

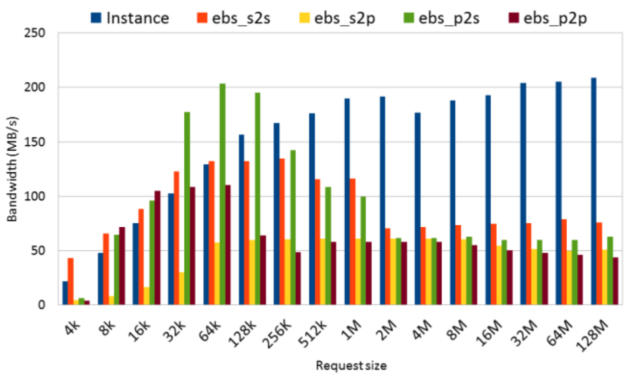

Figure 3. Sequential read bandwidth

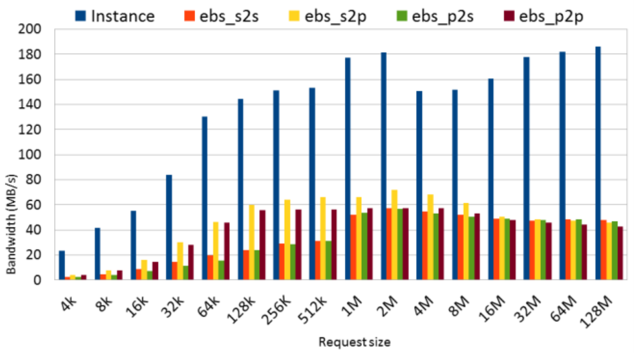

Figure 4. Sequential write bandwidth

Our interpretation of the results is that for EC2 the EBS volumes appear to be faster than the network. Hence, in the case of reads requesting large chunks of data becomes counter-productive because the responses are bottlenecked by the network leaving the drives idle. In the case of writes larger request sizes are better as there is no data to send back and there are less network overheads for larger request sizes.

A corollary of this is that for reads, provisioning IOPS on the EBS volumes is wasteful as the network is the bottleneck on the return path. More benefit is obtained by provisioning the network. On the other hand, for writes the drives cannot keep up with the network. This is likely a consequence of the EBS volumes being stored on SSD that have poor write performance as compared to read performance.

Noting that the peak write bandwidth of provisioned volumes is only marginally higher, we conduct further experiments without provisioned volumes.

4.1.2 X-stream baseline performance

For our tests we generated a synthetic undirected graph using the RMAT generator. We used 25 as the scaling factor making the number of vertices 2^{25} (32M) and the number of edges 2^{29} (512M). The I/O included reading in the edge list and writing out the updates for the gather phase as well as reading in the updates for the scatter phase. Since the entire vertex set fits into memory, only one partition was created and the vertices remained in memory for the entire experiment.

We ran BFS, Connected components (BFS forest) and Pagerank on the graph and the results are shown in Figures 5, 6 and 7. In addition to the storage configurations described above, we also experimented with increasing the number of volumes in the RAID array. The sequential access nature of X-Stream means that it can take advantage of the extra bandwidth offered by RAID arrays on physical machines. We therefore wanted to explore whether the same was also true in the cloud. The relative performance of EBS_P2S, EBS_S2S and Instance storage are consistent with the fio rest results. Instance storage performs the best followed by EBS_P2S and finally by EBS_S2S. **Moving from local instance storage to remote storage on EC2 affects the performance of graph processing by as much as 4X**.

Another conclusion from the results is that while RAID helps with local instance storage it has very little effect with remote storage. **The extra bandwidth afforded by sequential access to the RAID volumes is unavailable at the X-Stream end due the bandwidth limitations of the intervening network.** The improvements with RAID on local instance storage are consistent with X-Stream's performance on physical machines [1] that improvements for RAID in the region of 50% for two disks.

X-Stream has a key configuration parameter that dictates the size of requests made to storage when sequentially streaming data to or from it. By default (and in the results presented thus far) it is set to 16MB, as that was the optimum setting for physical machines with attached storage.

The fio test results however suggest that peak throughput is obtained using 1MB requests for sequential writes and 64KB requests for sequential reads. Therefore we decided to do another set of tests for each of the benchmarks with these I/O chunk sizes. The results

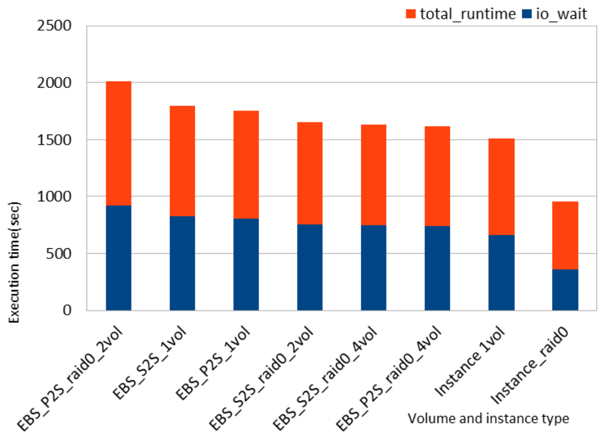

Figure 5. X-stream results for BFS

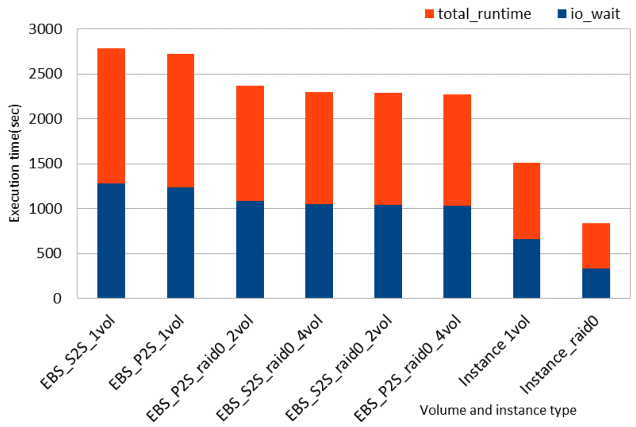

Figure 6. X-stream results for Connected Components

are shown in Figures 8, 9 and 10. We can conclude that a 1M I/O chunk size gives the best results with EC2 infrastructure as opposed to 16M with physical infrastructure. It is possible that even better performance might be possible by having different chunk sizes for the input and output paths. Unfortunately, X-Stream does not support this at the moment. Using X-Stream in the cloud would be a key motivator for adding such support.

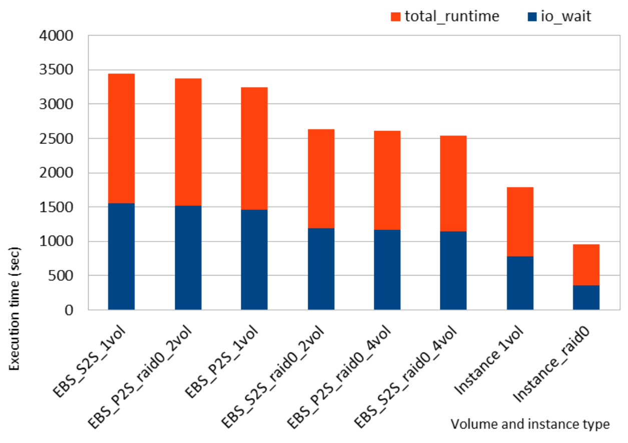

Figure 7. X-stream results for Pagerank

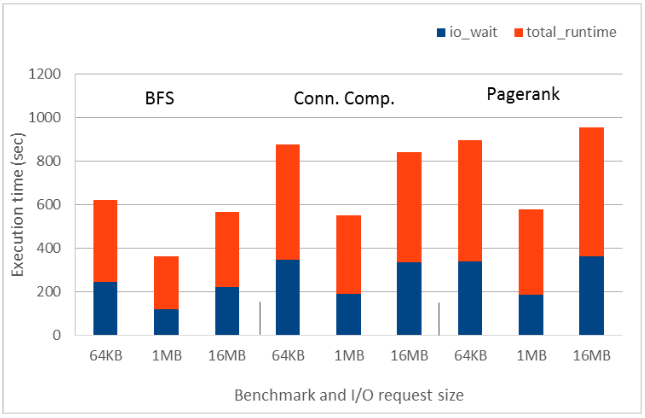

Figure 8. Results with variable request size on Instance store

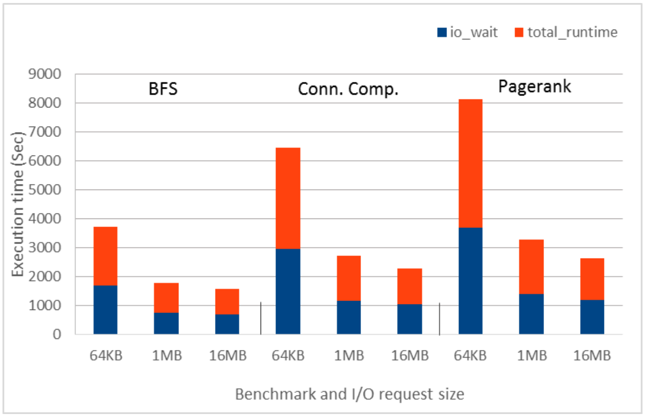

Figure 9. Results with variable request size on EBS_S2S

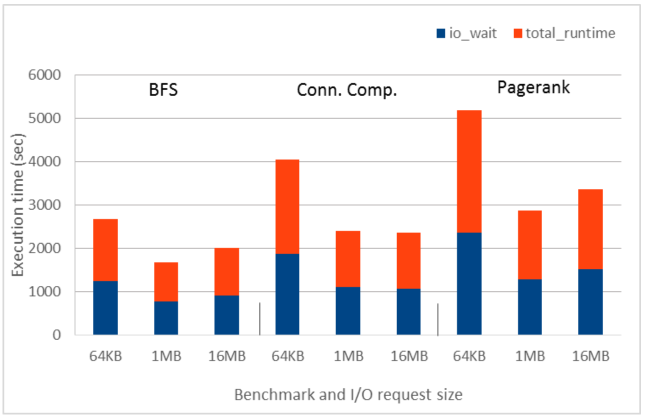

Figure 10. Results with variable request size on EBS_P2S

4.1.3 Compressed I/O

Provisioning the network comes with a trade-off of 9% in price increase for the instance for 12% of performance improvement in return.

Therefore we wanted to explore mitigating the bottleneck by storing and retrieving compressed sequences of edges and updates.

We analysed the gain with compressed I/O on both, EBS_S2S and EBS_P2S.

The sequential access nature of X-Stream makes such compression possible. We added compression support to the X-Stream codebase from [1]. We experimented with two different compression algorithms: zlib [5] and snappy [6]. Zlib provides somewhat better compression ratios in return for increased latency to decompress and compress blocks.

Graph name	Uncompressed	Zlib	Snappy
Synthetic graph	6GB	5.2GB(13.3%)	6.1GB(0%)
Twitter	17GB	11GB(35.3%)	14GB(17.6%)

Table 1. Zlib and snappy compression ratios for different graphs

In Table 1 we display the compression ratio of the synthetic graph we have used thus far in the paper as well as the Twitter [8] real world graph. The compression ratio is much better with the Twitter dataset due to the correspondence of vertex numbering with connectivity as the graph was generated by a crawler. In addition, the Twitter dataset has its edge-list sorted as an artefact of being generated from a compressed sparse row representation. The synthetic graph and the Twitter dataset therefore represent opposite ends of the spectrum in terms of compressibility of edge lists.

We ran experiments in this section with a 1M I/O chunk size, guided by our results from the previous section. We first consider the results displayed in Figures 11, 12 and 13 from executions on the synthetic graph. From these results, one may conclude the following:

It is possible for I/O bandwidth to exceed the capability of in-memory compression and decompression. This somewhat counter-intuitive result is because sequential accesses to SSDs used in EC2 is extremely fast, providing as much as 180 MB/s. On the other hand we have observed in separate tests that, zlib is unable to handle streams at more than 150 MB/s. This is why compression provides no benefit with instance stores. Snappy is faster than zlib and is able to keep up with the EBS volumes thereby providing benefit.

Benefits from compression can accrue even if the edge list is not compressible. Snappy provides benefits for synthetic graphs on EBS volumes even though the edge list is not compressible. This is because it is able to compress the updates reducing the time needed to write them out and read them back in.

Next, we consider breadth-first search over the real-world Twitter graph in Figure 14. This graph has a far better compression ratio for the edge list and therefore even the slower zlib is able to provide benefits for EBS volumes as the reduction in amount of data transferred more than makes up for the slower speed of zlib compression and decompression.

In general, compression is an effective technique to improving the performance of graph processing on EC2 using X-Stream. For our tests we were able to improve performance between 12%-30%.

Furthermore, we saw that when the compression is efficient, like with zlib on the twitter graph, the provisioned instance is cheaper by approximately 40%.

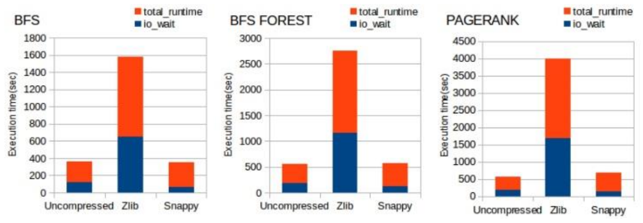

Figure 11. Compression on instance store

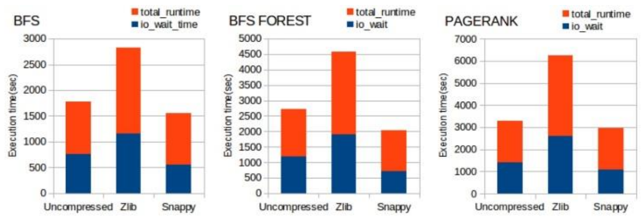

Figure 12. Compression on EBS_S2S

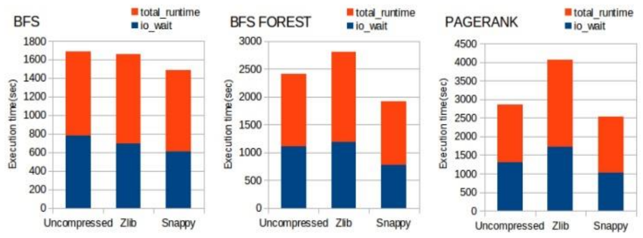

Figure 13. Compression on EBS_P2S

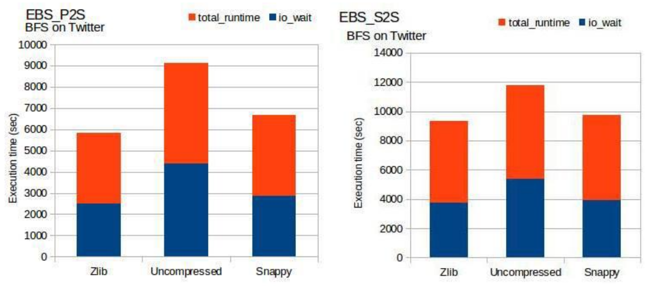

Figure 14. BFS on the twitter graph on EBS_S2S and EBS_P2S

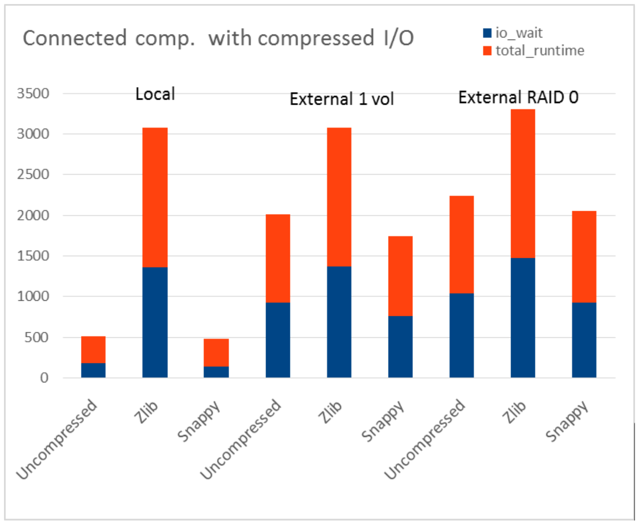

Figure 15. Windows Azure - X-stream execution times on different storage types and with different compression schemes on each type

5. Windows Azure

In order to generalize our findings we ran a subset of the tests performed on Amazon EC2 on Microsoft's cloud platform. The platform itself offers different options from EC2 and categorizes instances and storage in a slightly different manner. The first difference is that there is no provisioning of either network or IOPS but rather, they guarantee 500 IOPS for each attached disk. In this section we do not go into the specific details of the offered options but we chose the environment that is most compatible with what we had on Amazon.

The tests were run on Ubuntu 12.04 on Extra Large instances that offer 8 CPU cores and 16GB of RAM. Since this is larger than what we had on Amazon, we restricted X-stream to use the same amount of RAM as on EC2. We ran BFS, Connected Components and Pagerank on the following combinations of storage devices:

- Local storage
- One external disk of 20GB
- Two external disks with 20GB each striped in a software RAID-0 array

Before running X-stream, we ran the same fio tests as on EC2 in order to determine the best request size. The results are in agreement with the results on EC2 in the sense that the local storage is 6X faster for sequential reads and 2X faster for sequential writes. A difference was that for the external storage the peaks for reads and writes were for requests of 32MB. In order for our tests to be fair we chose this as our I/O chunk size on Azure.

Figures 15 illustrates the performance of X-stream on local and external storage as well as improvement that can be gained with compression.

As on EC2 the network is a limiting factor and the speedup gained by attaching new disks and striping them into RAID array is only marginal.

6. Conclusion

In this paper we have evaluated processing of large-scale graphs using X-Stream in the cloud and discussed the benefits and downsides of the cloud environment. Cloud storage is attractive due to the easy availability of large amounts of storage, something that is difficult to achieve with physical machines. One can therefore potentially scale the problem size tackled using the elasticity of the cloud.

Our most important conclusion is that the network is a serious bottleneck when accessing remote storage thereby limiting such scalability. This means that although cloud services can provide large amounts of storage for graphs, there is a penalty on performance due to the network to storage becoming a bottleneck.

For the specific case of EC2, we found a set of partial mitigations to this problem. Provisioning the network helps to improve performance by relieving the network bottleneck. A more cost-effective way of improving performance is through compression to reduce the amount of data moved on the network. We showed that it is necessary to use a compression algorithm with adequate performance to keep up with streaming bandwidths available on EC2.

A more effective solution for scaling graph processing in the cloud is to distribute X-Stream execution across multiple virtual machines thereby gaining aggregate bandwidth to storage. To this end we are engaged in the construction of a scale-out version of X-Stream. Unlike other distributed graph processing solutions such as Powergraph [9] this scale-out solution does not need to place the graph in main memory. We therefore would require far fewer virtual machines than Powergraph to process the same graph and – we expect- extend X-Stream's scope to terascale graphs (trillions of vertices and edges) in the cloud, something currently possible only on high-end supercomputers and massive clusters.

References

[1] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric Graph Processing Using Streaming Partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2013, pp. 472–488.

[2] https://www.windowsazure.com

[3] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/.

[4] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2012, pp. 31–46.

[5] http://zlib.net/.

[6] https://code.google.com/p/snappy/.

[7] http://freecode.com/projects/fio.

[8] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?," in *Proceedings of the 19th International Conference on World Wide Web*, New York, NY, USA, 2010, pp. 591–600.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2012, pp. 17–30.

.