DDPS A1 - Replication Study

Andreas Savva & Gareth Kok

1 Introduction

This is the first assignment for the course Distributed Data Processing Systems. The aim of this assignment is to reproduce real-world experiments conducted by top-tier researchers. We have selected the paper 'GraphX: Graph processing in a distributed dataflow framework' by Gonzalez et al [1] a 2014 paper about the Apache Spark GraphX library. The code used for this assignment is publicly available here `https://github.com/asavva3/GraphX-Reproducability` [1].

2 Background

The authors of the paper argue that the advantages of specialized graph processing systems can similarly be achieved using a modern general-purpose distributed data-flow system [1]. They introduce GraphX, an embedded graph processing system built ontop of Apache Spark. In addition to closely matching the performance of specific graph data processing systems, they argue that GraphX is simpler to use, provides fault tolerance and allows for a wider range of computations when compared to those systems.
As explained in the paper, graph computations are only a small part of the whole pipeline in an analytical process, and GraphX eliminated the need to support multiple systems or write data to files in order to move between systems. In addition, they do not provide fault tolerance, are not widely supported and result in unnecessary data movements which can be expensive. These are the reasons that motivate the creation and usage of a system such as GraphX.
On the other hand, distributed data processing systems do not perform well when executing iterative algorithms, early versions of them used on disk processing and did not allow for control on the data-partitioning. The last 2 limitations have been eliminated with the introduction of a new generation of distributed data processing systems such as Spark.
GraphX is built on top of Apache Spark as a library and uses the Pregel abstraction. It allows for the use of both unstructured and structured data to be used as collections and graphs.

3 Experiments

The authors evaluate the performance of GraphX against various systems, including GraphLab, Giraph, optimized Spark, and naive Spark. Comparing the performance of the page-rank algorithm as well as the connected components algorithm on the popular datasets twitter-2010 and uk-2007-05.

[1]https://github.com/asavva3/GraphX-Reproducability.git

Both graphs used by the authors are directed graphs with the former consisting of 1,468,365,182 edges and 41,652,230 vertices and the latter consisting of 3,738,733,648 edges and 105,896,555 vertices. In addition they tested the scalability of GraphX, the effects of partitioning on communication and fault tolerance.

3.1 Chosen Experiments

We had to choose two experiments to reproduce in order to see if we could achieve similar results with the authors. Our two experiments of choice were the System Performance Comparison and the Scaling experiments.
The first experiment was chosen because we believe that on of the most important aspects of the system is its performance and its evaluation is crucial. We chose to compare GraphX against naive Spark for the page-rank and Connected Components algorithms. Naive Spark was chosen as the second system since proving that GraphX has substantially better performance than implementing the algorithms on a bare distributed data processing system proves its validity as an alternative to the specialized graph processing systems.
Scalability is one of the most important aspects of systems like these, which have to process large amounts of data. If a distributed data processing system doesn't scale then it defeats the purpose of using it. That is why we chose to perform the scalability experiment as the authors did using the page-rank algorithm.
We faced issues unpacking the twitter-2010 dataset onto the server and transferring it to the HDFS partition. Using the twitter dataset compressed never imported in our programs. When we unpacked the dataset using gzip, we tried transferring it to HDFS but it always failed with error code showing unexpected EOF. We also tried importing the unpacked dataset from the dfs file system but that failed as well. Furthermore we were unable to find the uk-2007-05 dataset in a usable format. The only uk dataset we found was in a .graph file in BV graph format and we couldn't import it as it was because it wasn't a recognized format for spark and GraphX. As a result, we opted for using large directed graphs available on the SNAP dataset collection web-page [2]. All graphs we use could not be processed on a local machine in a reasonable amount of time or resulted in memory exceptions, hence we decided that they would be suitable replacements to the networks utilised by the original authors. Figure 1 displays the datasets we used for evaluation of the various systems.

Network	Nodes	Edges
soc-LiveJournal1	4,847,571	68,993,773
soc-Pokec	1,632,803	30,622,564
cit-Patents	3,774,768	16,518,948

Table 1: Networks available through the SNAP dataset collection [2]

In order to evaluate the performance of the various systems we compare the run-time of the page-rank and connected components algorithms using a number of different cluster settings. Both page-rank and the calculation of connected components are iterative graph algorithms, with page-rank measuring the level of importance of nodes within a graph and the number of connected components is a measure of reachability. With directed graphs, the measure of connected components refers to the number of strongly connected components, which is a sub-graph in which all nodes are reachable from one another.

3.2 DAS-5 System

The authors of the paper listed Apache Spark 0.9.1 for their experiments. They run their experiments on Amazon EC2 16 m2.4xlarge nodes, each having 8 vcores, 68GB memory and two hard disks. We used the latest available release for Apache Spark which at the time of testing was 3.2.0 which supports Apache Hadoop 3.0 and later. We also used Apache Hadoop version 3.3.1 to setup a yarn cluster and HDFS.
All experiments are conducted on the DAS-5 systen using 24 large worker nodes. With each node consisting of 8-cores, 64 GB of memory, and 2 4TB hard disks. The cluster runs CentOS linux.
Using the Apache Hadoop we setup yarn to be our resource manager. In the core-site.xml configuration file we enter our default file system URI, which in our case in HDFS , along with the master URL.

Property	**Value**
fs.default.name	hdfs://master-url:9000

Table 2: core-site.xml

In the hdfs-site.xml we set the mount point for our namenode and datanode directory. We also set replication to 1. Further more we edit yarn-site.xml, where we will enter the address of our resource manager, amount of physical memory available in the containers(yarn.nodemanager.resource.memory-mb), which for our case was set to 61440MB leaving some memory for other tasks that might be running. We also set the amount of v-cores to 6, leaving 2 for any overheads and the maximum allocation to every container to 10GB.

Property	**Value**
yarn.resourcemanager.hostname	master-url
yarn.nodemanager.resource.memory-mb	61440
yarn.nodemanager.resource.cpu-vcores	6
yarn.scheduler.maximum-allocation-mb	10240

Table 3: yarn-site.xml

Lastly we edit the mapred-site.xml where we enter our hadoop path for map reduce env. To indicate the nodes where our workers are, we edit the workers file, in which we add the addresses of our worker nodes, using the addresses 10.149.1.*. These configurations were chosen to take full advantage of our cluster computers. To setup spark we had to edit the spark-env.sh configuration and add our spark master host, hadoop configuration directory and yarn configuration directory. We also added our workers in the workers configuration file of spark.
To conduct our experiments we moved our datasets to HDFS and run the scala jar files we created. We used the spark-submit function, set our master to be yarn and deploy mode to cluster. We also set the number of executors, memory and vcores. The numbers where found after experimentation and reading to find the suggested ones. We believe that the variables used here are good for our

setup, and do not results in too large or too small executors. The number of executors depends on the number of workers and we aim for 3 executors per worker except the application master (AM) container. We used 8GB of ram for each executor, and 2 vcores.

3.3 Programs

For the experiments, we had to run the Connected Components and page-rank algorithms on GraphX and naive page-rank. The GraphX git repository and documentation have examples for how to run the two algorithms on the system. Since the authors haven't made available the exact code they used, we used these examples and just made slight changes. For the naive Spark implementation, there is also an example available for page-rank that does not use GraphX but no example for Connected Components. Again, for naive Spark, we used the example with small modifications. Unfortunately we were not able to find any Connected Components implementations. For our implementations, we wanted to get the run times only for running the algorithms, so we excluded the part of importing the data and creating the graph. In order to get the performance results for GraphX, we used the command spark.time, which gives the run time for an operation miliseconds,for the two algorithms on the graph. For the naive Spark implementation we used System.nanoTime, which gave us the time in nanoseconds. Each experiment was run 10 times in order to get a feel of the average performance of the various systems.

4 Results

Figures 1 shows the runtime performance on the page-rank algorithm implemented in GraphX when scaling the number of nodes in a cluster. In general for all of the networks we see that scaling from 2 to 4 nodes results in a decrease in runtime, however scaling further to 6 nodes results in either a decrease in performance or little to no change. Similarly in figure 2 we see that scaling from 2 to 4 nodes results in a decrease in runtime for the connected components algorithm implemented in GraphX, however scaling further results in insignificant changes. Figure 3 shows the performance of the naive Spark page-rank implementation on the various networks against that of GraphX page-rank implementation on a 6 node cluster. We see that GraphX outperforms naive Spark on all datasets, matching the conclusions drawn by the original authors.

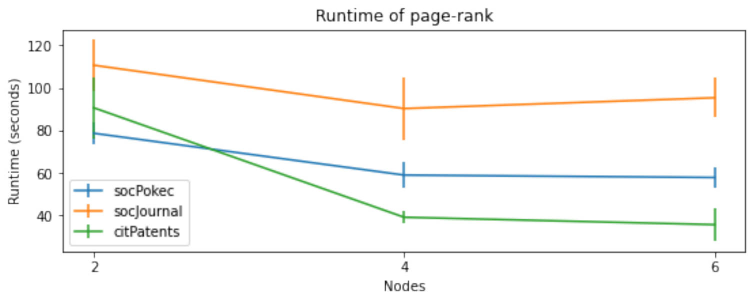

Figure 1: Runtime of the page-rank algorithm with standard error across 20 iterations.

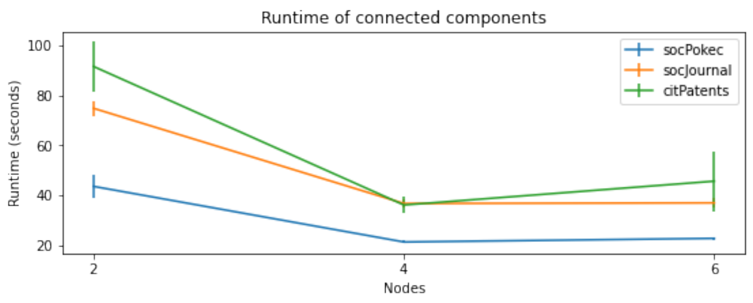

Figure 2: Runtime of the connected components algorithm with standard error across 20 iterations.

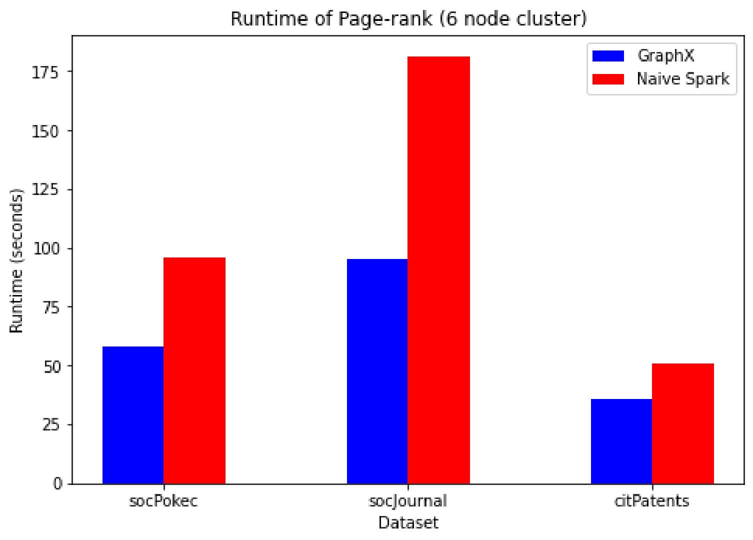

Figure 3: Runtime of the page-rank algorithm with standard error across 20 iterations.

5 Discussion & Conclusion

While we were unable to evaluate the performance of the two systems using the same datasets as the original authors, we still show that the GraphX system outperforms the naive Spark implementation. we can still see that GraphX performs notably better than naive Spark and we believe that if we could perform the experiments with larger datasets as the ones the authors used, we would have similar results as the authors and show that naive Spark wouldn't be able to handle those computations in a reasonable amount of time.
Furthermore our results for scalability show that the datasets we used were small enough that we could process them using four nodes. We see that the GraphX system scales well from two to four nodes on all of our datasets but we do not get any improvements when moving to six nodes. We believe that this is because at that point, the communications cost outweigh the computation. From two nodes to four nodes we see about 1.5x performance improvement. We believe that this is in line with what the authors have discussed in their paper where they increase the machines by a factor of four and observe a 3x speedup. If our dataset was in the same order as theirs, we would see similar scaling.

In future research we would like to measure the actual communication between the nodes in the cluster, in order to be sure about our hypothesis about the communication when scaling to 6 nodes. We should also mention here that we tried to use Apache Giraph for our experimentation about performance. We followed the documentation and used the versions that were mentioned but we could not get the examples to run on the cluster. In addition, when searching for GraphLab, we could not find any usable documentation to use in order to setup for the cluster.

6 Task Distribution

Task	**Andreas Savva**	**Gareth Kok**
Document	50	50
Code	40	60
Cluster Setup	60	40
Planning	50	50
Experimentation	60	40
Research	50	50

Table 4: Task distribution %.

References

[1] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, October 2014. USENIX Association.

[2] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.