Report Master Basis Module, Spring 2014
Spark Implementation of the Centroid Decomposition Method

Alessandro De Carli, 10-751-717

July 8, 2014

1 Introduction

In our digital era the amount of produced information is tremendously growing every year [4]. There's a need for fast and scalable algorithms to analyze this data and understand the correlations between the data. Matrix decomposition techniques such as the Centroid Decomposition allow the analyze the correlation between data. The most efficient algorithm to compute the Centroid Decomposition has qudratic runtime complexity and thus does not scale to large datasets. The aim of this thesis is to provide a distributable implementation of the Centroid Decomposition technique.

This report introduces a distributed implementation of the Centroid Decomposition using Spark platform. Section 2 introduces the background of this work. Section 3 describes the proposed implementation. Section 4 shows the results of running the implementation on a cluster. Section 5 summarizes this report and discusses the limitation of this work and some further optimizations of the current implementation.

2 Background

2.1 Scalable Sign Vector

The Centroid Decomposition (CD) is a matrix decomposition technique that decomposes an input matrix into the product of two matrices as described in Algorithm 1. The decomposition is performed iteratively and the input matrix is updated at each iteration.

At the core of the proposed algorithm is the method *FindSignVector()* that finds the sign vector. The original CD algorithm computes the sign vector in exponential runtime and linear space. In [2], the authors introduce an algorithm to compute the sign vector in quadratic runtime and space complexity. In [5, 6] the authors describe a greedy approach the compute the sign vector in linear space complexity and quadratic runtime complexity. The proposed solution is described in Algorithm 2.

Algorithm 2 iteratively computes the sign vector based on the result of previous itertations. In the worst case, the algorithm requires to update the sign of n elements

Algorithm 1: CD($\mathbf{X}$, n, m)

Input: $n \times m$ matrix $\mathbf{X}$
Output: $\mathbf{L}$, $\mathbf{R}$

$L = R = []$;
for $i = 1$ **to** m **do**
  $Z = FindSignVector(\mathbf{X}, n, m)$;
  $C_{*i} = \mathbf{X}^T \cdot Z$;
  $R_{*i} = \frac{C_{*i}}{\|C_{*i}\|}$;
  $\mathbf{R} = Append(\mathbf{R}, R_{*i})$;
  $L_{*i} = \mathbf{X} \cdot R_{*i}$;
  $\mathbf{L} = Append(\mathbf{L}, L_{*i})$;
  $\mathbf{X} := \mathbf{X} - L_{*i} \cdot R_{*i}^T$;
return $\mathbf{L}$, $\mathbf{R}$

Algorithm 2: SSV($\mathbf{X}$, n, m)

Input: $n \times m$ matrix $\mathbf{X}$
Output: maximizing sign vector $Z^T = [z_1, \ldots, z_n]$

$pos = 0$;
repeat
  `// Change sign`
  if $pos = 0$ **then** $Z^T = [1, \ldots, 1]$;
  else change the sign of z_{pos};
  `// Determine` S `and` V
  $S = \sum_{i=1}^{n} (z_i \times (X_{i*})^T)$;
  $V = []$;
  for $i = 1$ **to** n **do**
    $v_i = z_i \times (z_i \times X_{i*} \cdot S - X_{i*} \cdot (X_{i*})^T)$;
    Insert v_i in V;
  `// Search next element`
  $val = 0, pos = 0$;
  for $i = 1$ **to** n **do**
    if $(z_i \times v_i < 0)$ **then**
      if $|v_i| > val$ **then**
        $val = v_i$;
        $pos = i$;
until $pos = 0$;
return Z;

yielding to a time complexity of $O(n^2)$ where n is the number of rows of the input matrix. This algorithm is optimized for input matrices that have the number of rows much bigger that the number of columns. While the linear space complexity makes the algorithm scalable in terms of space, the bottleneck now lies on its quadratic runtime complexity, which makes it slow for large datasets.

In summary, we are dealing with an algorithm that is iterative and has quadratic runtime complexity. Therefore, we propose in this thesis to implement the Centroid Decomposition under a distributed platform.

2.2 Hadoop

Apache Hadoop is a distributed framework commonly used to to make algorithms scalable to large datasets. This framework gives the user an abstraction allowing the execution of tasks in parallel on a cluster of nodes . Hadoop uses a file system named Hadoop File System (HDFS) that is fully distributed and thus, has no bottleneck.

2.2.1 Hadoop File System

The Hadoop File System is one of the modules included in the Apache Hadoop project. The module allows the user to build a distributed filesystem over a cluster. This abstraction gives the developer the possibility to write programs that read and write files without worrying about chunking and distributing large files on the cluster. In fact, HDFS chunks a large file into blocks and distributes them on the nodes of the cluster. Furthermore, HDFS replicates a block on multiple nodes in order to get fault tolerance (in case a machine leaves the cluster) [1].

2.2.2 Map Reduce

Apache Hadoop works with a core concept called MapReduce. MapReduce allows to write parralelizable code by specifying two functions: map and reduce. The map function "maps" a given key/value pair onto one or multiple other key/value pairs [3]. The reduce function collects multiple key/value pairs of the same key and outputs one or none key/value pairs.

2.2.3 Iterative Tasks

Apache Hadoop platform is not suitable for iterative tasks. In fact, while Apache Hadoop provides an abstraction for accessing computational resources, it lacks abstractions that allow access to the clusters' main memory [8]. The only way a user can share results among multiple Map Reduce Tasks is by writing them to the distributed filesystem. In order to preserve the fault tolerance, HDFS replicates the written file among the nodes, which results in an overhead in disk I/O. In order to overcome this limitation, a new platform called Spark has been introduced.

2.3 Spark

Zaharia et al.[9] introduce a new distributed framework for iterative algorithms named Spark. The latter copes with the limitation of Hadoop to access the memory of the nodes. Spark provides a computational framework that gives an abstraction to access the distributed main memory using Resilient Distributed Datasets (RDD's).

2.4 Resilient Distributed Datasets

Ignoring the main memory resource, will result in an overhead of disk I/O, which is not suitable for iterative tasks. This is where the concept of Resilient Distributed Datasets comes in place. RDD's are partitioned collections of objects which are distributed in a cluster's main memory, they are resilient which means they are fault tolerant. RDD's handle the map and reduce operations as chainable coarsed grained transformations, meaning that the data gets read once from the HDFS, but all subsequent transformations will take place in-memory. RDD's are lazy, the transformation chain would only be executed, if an actual result is requested from them [8].

While such an approach requires an appropriate remodelling of data structures in order to be compatible with RDD's, it scales well for iterative algorithms such as the SSV.

3 Implementation

3.1 Data Representation

The most challenging part of this work is to represent the data structures in such way that they can be efficiently processed by Spark. CD algorithm takes as input a matrix and delivers two matrices as output. The two data structures that we have to deal with are matrices and vectors. An RDD represents a collection of key-value pairs distributed on the cluster, this means that vectors are trivial in their representation: we take the vector's position index as key and the value as value. For matrices however finding an efficient and scalable representation is less intuitive. In this thesis three different representations have been implemented. In what follows, the three representations are explained and iluustrated with the according code snippet. All representations use the same input file format, which is a simple space separated file following this structure:

```
# columnIndex rowIndex value
0 0 2
0 1 0
0 2 -4
1 0 -2
1 1 3
1 2 2
```

3.1.1 Indexed List Representation

The first representation is based on the idea of storing the columns of the input matrix as values (in form of a list) and the position index of the column as keys. This representation was used since many calculations require the use of an entire column (X_{i*} (cf. Algorithm 2. This solution allows to efficiently perform these calculations on a single machine, without the need to fetch and aggregate further values. This approach works well for a local computations where the value order of the list entries is fixed, but fails when the computation gets performed (as it should be) in a distributed manner. The reason for this, is that the data is read blockwise from the HDFS thus we cannot ensure that a node keeps the list in a correct order without having the overhead of checking the values position in every insert. The second drawback of such approach is that since $n >> m$, then using an indexed list representation on each node requires the storage of the complete column. Therefore, for an input matrix containing few number of columns, e.g., three, the computation can be distributed only on three machines, which makes it not scalable.

```
JavaPairRDD<Integer, List<Double>> rowKeyMatrix = ...
JavaPairRDD<Integer, List<Double>> columnKeyMatrix = ...
```

3.1.2 Tuple-Index Representation

A second representation of the data structures named Tuple-Index representation was implemented. This representation is the complete opposite of the first one. The main idea is to use a tuple containing two indices (one for the row position and one for the column position). This tuple represents the key and the value located at that coordinate in the input matrix is the value. Since Apache Spark works with the MapReduce framework, this representation fails in case of using an RDD that needs different key structures. As an example, the line five of Algorithm 2 is a multiplication of a matrix column with the Z Vector. Since the vector has only one index and the matrix has two, performing a join on these different keys is not supported by Apache Spark.

```
JavaPairRDD<Tuple2<Integer, Integer>, Double> matrix = ...
```

3.1.3 Joinable-Index Representation

The third data structure representation copes with the limitations of the two previous representations. The idea is to create two different RDD's for the same matrix. The first RDD takes the column position index as key and has as value a tuple holding the row index and the actual value. The second RDD flips the indicies, meaning the row position index will be the key. In case a join on column positions is peformed, the first RDD is taken, In case a join on row position is required, the second RDD is taken. This approach is completely distributable, since the RDD's entries consists out of very small objects and the transformations can be easily chained.

```
JavaPairRDD<Integer, Tuple2<Integer, Double>> rowKeyMatrix = ...
JavaPairRDD<Integer, Tuple2<Integer, Double>> columnKeyMatrix = ...
```

3.2 Matrix Operations Description

Using RDD allows to create distributable algorithms. The drawback on the other hand is that the operations possible on these data structures are limited. That's why very trivial calculations need to be designed in a way that they can be computed only by using the limited operations provided by the framework's API. In the following paragraphs, the most important operations needed for the implementation are explained:

Matrix multiplication: matrix multiplication is the main operation used in the implemented algorithm. By using the Joinable-Index representation, the distributed matrix multiplication is executed by first performing a join on the values that need to be multiplied. This join maps both values multiplied to a given key. The result is an RDD that has partial results of the multiplications mapped to a key. Then, the RDD will be reduced, where an aggregation of the value (sum) based on the key is performed. The last transformation already provides an RDD representing the matrix multiplication result.

Matrix subtraction: Two matrices are subtracted one from the other by first mapping the subtracting matrix with a "-1" multiplication on each value. Then, both matrices can be joined value by value and reduced by using a simple sum as aggregation. This approach addresses the reduce problem of the subtraction that results when the values are aggregated, since we cannot determine which value is the subtractor.

Find maximizing sign vector: Algorithm 2 iterates until finding the maximizing sign vector. Due to the fact that every iteration depends on the maximizing sign vector, then the latter needs to be distributed among the nodes in the cluster during the calculation. Apache Spark provides different ways to share variables among the nodes:

- RDD → key-value pairs only, suitable but needs redesign since it can only be
- Broadcast variable → read only, not suitable since the vector changes with each iteration.
- Accumulator variable → (limited) write only, not suitable since the value is needed for the next iteration.

Based on the limitations of broadcast and accumulator variable, this implementation represents the Z vector as an RDD, with each iteration adding the transformation to its chain.

4 Results

The proposed implementation is executed on the ifi's Tentacle cluster at the university of Zurich. The cluster consists of 93 nodes each node has 8GB ram and a quadcore. The following tests were deployed in "yarn-cluster" mode. The shown results are based on the monitoring dashboard spark provides. The input dataset is a matrix with 10 columns and 100'000 rows.

4.1 Task parallelization

Task distribution is very limited due to the fact that the SSV algorithm provides very little parallelization potential. The calculations that can be parallelized are all very short. Every parallelization on a cluster leads to a coordination overhead, in plus since Spark is fault tolerent, data is replicated which is again a further overhead.

```
1 NODE
======
100000x10 Matrix Loaded
Loading Elapsed: 7814254000ns
Iteration Elapsed: 8s

10 NODES
========
100000x10 Matrix Loaded
Loading Elapsed: 16108332000ns
Iteration Elapsed: 18s
```

Figure 1: Execution Time Log

Figure 1 shows that the calculations are performed faster if a single node is used. This can be explained by the fact that the overhead added by distribution is bigger than the advantage gained of parallelizing the short calculations.

Figure 2 shows how the tasks are distributed in a cluster of 10 machines. We can see that the distribution is highly unbalanced. This is a further observation demonstrating how limited the SSV algorithm can be distributed.

Executors (11)

Memory: 991.7 MB Used (24.1 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Shuffle Read	Shuffle Write
1	imac-stud-011.ifi.uzh.ch:54297	15	15.2 MB / 2.3 GB	0.0 B	0	0	25	25	16.7 s	135.7 MB	10.9 MB
10	imac-stud-091.ifi.uzh.ch:58809	15	99.5 MB / 2.3 GB	0.0 B	0	0	47	47	27.7 s	141.0 MB	29.0 MB
2	imac-stud-079.ifi.uzh.ch:57301	12	99.5 MB / 2.3 GB	0.0 B	0	0	21	21	17.8 s	51.0 MB	38.6 MB
3	imac-stud-001.ifi.uzh.ch:54035	14	130.0 MB / 2.3 GB	0.0 B	0	0	96	96	46.3 s	206.6 MB	106.1 MB
4	imac-stud-071.ifi.uzh.ch:60943	12	22.8 MB / 2.3 GB	0.0 B	1	0	19	20	14.1 s	72.7 MB	19.8 MB
5	imac-stud-044.ifi.uzh.ch:53232	11	7.6 MB / 2.3 GB	0.0 B	0	0	20	20	16.5 s	110.9 MB	10.9 MB
6	imac-stud-075.ifi.uzh.ch:57263	74	579.0 MB / 2.3 GB	0.0 B	0	0	162	162	51.1 s	8.5 MB	988.3 MB
7	imac-stud-070.ifi.uzh.ch:55084	10	7.6 MB / 2.3 GB	0.0 B	0	0	16	16	13.4 s	101.5 MB	8.2 MB
8	imac-stud-012.ifi.uzh.ch:50916	19	15.2 MB / 2.3 GB	0.0 B	0	0	30	30	24.5 s	159.7 MB	20.4 MB
9	imac-stud-014.ifi.uzh.ch:61996	10	15.2 MB / 2.3 GB	0.0 B	0	0	13	13	7.9 s	35.4 MB	2.7 MB
<driver>	imac-stud-102.ifi.uzh.ch:50597	0	0.0 B / 1092.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B

Figure 2: Executor Dashboard during SCD of 10 nodes

4.2 Lineage Overflow

RDD's were originally introduced for coarse transformations [8]. As shown in paragraph 3.2 the Z vector is modelled as an RDD that keeps adding transformation with each iteration. Since the transformation are very small (1 value per iteration) the choice of using an RDD is not optimal. Each iteration adds a new transformation, which leads to a lineage overflow for large datasets.

Executors (2)

Memory: 1044.4 MB Used (3.4 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Shuffle Read	Shuffle Write
1	imac-stud-074.ifi.uzh.ch:53708	195	1044.4 MB / 2.3 GB	0.0 B	1	0	458	459	2.9 m	0.0 B	1256.3 MB
<driver>	imac-stud-012.ifi.uzh.ch:50761	0	0.0 B / 1092.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B

Figure 3: Executor Dashboard during SCD of 1 nodes

Figure 3 shows the growth of memory usage as more iterations are performed. If the algorithm is ran on a single machine, the overflow is reached after 7 minutes. If it is ran on a cluster of 10 nodes, the overflow is reached after 4 minutes. The reason for this is that many more transformations can be attached to the lineage if the task is distributed, but every node that wants to perform the next transformation needs to know the complete lineage, which leads to a faster overflow.

5 Conclusion

In this thesis, we have provided a distributed implementation of the Centroid Decomposition technique proposed in [6] using Spark framework. The latter provides a framework to allow distributed in-memory computations on clusters, which comes in handy for iterative algorithm such as for SSV algorithm.

The major findings of this thesis are a representation of the data structures that allow a correct calculation of matrices. Using the proposed "Joinable-Index" representation together with a distributed filesystem like HDFS, allow to perform matrix calculations in a fully distributed manner.

The main limitation of the proposed implementation is that the tasks in the algorithm are hard to parallelize. As shown in Section 4.1, the overhead of distribution cannot overcome the benefit of the distributed computation. The second faced limitation is provided by the way RDD's work, with many iterations lineage overflows are quickly reached. RDD's should be used for coarse grained transformations and not small transformations as used in for the sign vector Z.

Future work should focus on the optimisation of task distribution since the tasks are not distributed evenly among the workers in the cluster, the current bottleneck lies in the poor paralellisability of the SSV algorithm. Also due to the lineage overflows another possible optimization is to distribute only parts of the algorithm that can be parrallelized and compute the rest of the algorithm on a single master node, while such a combined approach could provide an advantage in terms of performance, this would introduce a central bottleneck to the master node.

Bibliography

[1] Milind A. Bhandarkar. Mapreduce programming with apache hadoop. In *IPDPS*, page 1, 2010.

[2] M.T. Chu and R.E. Funderlic. The centroid decomposition: Relationships between discrete variational decompositions and svds. *SIAM J. Matrix Anal. Appl.*, 23(4):1025–1044, 2001.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, April 2011.

[5] Mourad Khayati, Michael H. Böhlen, and Johann Gamper. Scalable centroid decomposition. In *Technical Report*, 2013.

[6] Mourad Khayati, Michael H. Böhlen, and Johann Gamper. Memory-efficient centroid decomposition for long time series. In *ICDE*, pages 100–111, 2014.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, page 1–10. IEEE, 2010.

[8] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[9] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.