Computer Science and Information Systems 19(1):117–139 https://doi.org/10.2298/CSIS200531039S

Scaling industrial applications for the Big Data era

Davor Šutić and Ervin Varga

Faculty of Technical Sciences, Trg D. Obradovića 6,
21000 Novi Sad, Serbia
{sutic, evarga}@uns.ac.rs

Abstract. Industrial applications tend to rely increasingly on large datasets for regular operations. In order to facilitate that need, we unite the increasingly available hardware resources with fundamental problems found in classical algorithms. We show solutions to the following problems: power flow and island detection in power networks, and the more general graph sparsification. At their core lie respectively algorithms for solving systems of linear equations, graph connectivity and matrix multiplication, and spectral sparsification of graphs, which are applicable on their own to a far greater spectrum of problems. The novelty of our approach lies in developing the first open source and distributed solutions, capable of handling large datasets. Such solutions constitute a toolkit, which, aside from the initial purpose, can be used for the development of unrelated applications and for educational purposes in the study of distributed algorithms.

Keywords: distributed computing, big data, smart grid.

1. Introduction

Large industrial complexes, e.g. utilities or factories, rely on timely and accurate telemetry data as well as layers of redundancies that keep the production going even in the case of a failure. Up until the recent past, applications behind their operation were focused on a relatively small and static amount of data. Once set up, the infrastructure needed periodic maintenance, but had little demand for a change of scale.

With the advent of Smart Grid infrastructures this concept gradually changed. It became common to change the scale of the operation by adding more customers or introducing smart devices into the ecostructure. The increasing amount of required data demands puts also an additional strain on the available computational power. The algorithms used for analyzing aspects of the operation are usually non-trivial and thus the large amounts of data challenge their applicability. However, with the changing scope of the applications, the preferred infrastructure shifted to larger distributed systems, whether in the cloud or not.

In this paper, we present three projects that illustrate how hard industrial computational problems are solved on large datasets. The unifying factors of all three solutions are a common framework and distributed environment. They rely on Apache Spark to provide a common infrastructure in order to share a communication foundation and facilitate comparison between them. They also primarily target large datasets, i.e.

scales that would be hard for non-distributed applications to compute in a reasonable time.

The first two projects can be put under an umbrella of smart grid power analysis. They introduce support for the power flow and contingency analysis functions. The power flow analysis is performed using the Newton-Raphson method, while the contingency analysis is performed in two distinct approaches, the network connectivity state is assessed through the analysis of the graph constructed from the connected components of the network and through the binary multiplication of Boolean matrices.

The third addresses a missing utility in graph processing algorithms. The complexity of processing a graph quickly increases with its size, so it would be beneficial to decrease the complexity of the graph while maintaining its mathematical properties. Here, we provide a reusable distributed spectral graph sparsification solution. Reducing the number of vertices is usually not desired due to their semantic importance. Luckily, real-world graphs tend to have more edges than vertices, so reducing the number of edges both reduces the size of the problem and doesn't affect the semantic of the dataset.

This paper shows the applicability of the Apache Spark framework to industrial applications. The open source [15][16] solutions herein are the first of their kind both in handling large datasets in a distributed manner and in the map reduce paradigm, which also motivated the choice of the problems.

The paper is outlined as follows: The next section addresses related work. Section 3 presents the power flow problem and outlines our solution. Section 4 presents the island detection problem and outlines our solution. Section 5 presents the spectral graph sparsification problem and outlines our solution. The following section details the experimental setup and its results. Finally, we conclude the paper, by presenting a short overview of the contributions and an outlook for future research.

2. Related work

Being a distributed data processing engine, Apache Spark [1] has since its introduction found a wide range of users. Applications include various disciplines where the problem can be reduced to analyzing large amounts of data, like genomic analysis [2] and specialized mathematical methods for matrix computation [3].

The power flow problem was stated decades ago [4], however, once the Newton-Raphson method was introduced [5], only one other solution method was developed [6]. Improvements have mainly been directed towards the benefit of mathematical apparatus, pre-dominantly focused on matrix algebra, used by the solutions methods.

The island detection problem, that is an integral part of contingency analysis, was prominently approached in [7], by using a network connectivity matrix in conjunction with Boolean algebra. Yet, in the paper discussion J. L. Marinho et al. challenge the solution by calling it "unnecessarily complex" when compared to graph analysis approach. The authors' rebuttal accentuates the advantages of their solution and state that in their experience the proposed graph-search algorithms were not faster. This exchange is important as it sets the main directions of island detection research early on, towards matrix analysis improvement [10], [8], or towards more advanced topology analysis [9]. Finally, it constitutes the main incentive for us to compare both approaches.

Currently, the most prominent open source power analysis tools are based on MATLAB [11], [12]. From the Java based tools, it is worth to mention DCOPFJ [13] and InterPSS [14]. Yet, what all these tools lack is a distributed solving mechanism.

The notion of spectral graph sparsification, that is relevant to this paper, was introduced by Spielman and Teng [17] and focuses on the spectral similarity between a graph and its sparsifiers. Their main result is the proof that every graph has a near-linear sized spectral sparsifier that can be computed in near-linear time. However, as they state, the powers of logarithms and the constants in their achieved upper bound are too large to be of practical importance, but their goal was in any case to prove that such sparsifiers exists and not the optimization of the process. In this paper we focus on exactly that practical aspect and show a solution that can successfully sparsify large graphs with relatively modest resources in a practically acceptable amount of time. Further, [17] is the second in a series of three papers [18][19] with the ultimate goal to develop efficient methods for solving linear systems in symmetric, weakly diagonally dominant matrices. In terms of spectral graph theory, that is important for finding the eigenvalues of Laplacian matrices. Our paper uses the graph partitioning algorithms presented in [18] to support the sparsification effort. The quest for the eigenvalues of a graph's Laplacian [19] is beyond the scope of this paper, however, it constitutes a logical continuation of the herein described approach. Another implementation was done by Perraudin et al. [35] in order to extend an existing open-source graph signal processing toolbox. Being written in MATLAB, the solution is inherently non-distributed and highly specialized, which limits its efficacy when managing large datasets. Thus, the approach fundamentally differs to our solution.

Spielman and Teng [17], and Spielman and Srivastava [20] inspired further research in finding a distributed approach. Koutis and Xu [21] introduce a theoretical algorithm for spectral sparsification. Their work focuses on the use of weighted spanners. The computation of which is, however, complex and expensive in terms of resources. That is why we choose to build upon the original algorithm [17], which carries an arguably more intuitive set theory mindset. Unfortunately, Koutis and Xu [21] didn't provide any benchmark, so it is hard to compare our solution to their approach. Similarly, Sun and Zanetti [22] approach the sparsification problem from a clustering perspective avoiding spectral methods. Their experiments focus on the functionality of their algorithm. The datasets they use are magnitudes smaller than ours, so a direct comparison is hard to make. However, they argue that spectral sparsification methods are complex and thus unsuitable for the distributed setting, which we disprove in this paper.

Šutić and Varga [23] expanded the Apache Spark GraphX library [1][25] with the notion of distributed spectral graphs and basic spectral analysis operations. Some of them, e.g. the Laplacian matrix calculation, are used in this paper, but otherwise are the contributions of this work a logical extension of the existing framework.

Spectral graph sparsification has important applications. By reducing the number of edges in graphs, while preserving the properties, methods that were previously too expensive to use, become tractable and applicable. A particularly illustrative example is the problem setting of the work by Zhao et al. [26]. Namely, they propose a new method based on spectral graph sparsification for the modelling and simulation of large power delivery networks. They look back at various methods for achieving that and conclude that none can rise to the challenges of the complexity of contemporary power grids while simultaneously keeping the required accuracy. The proposed solution itself uses

spectral sparsification to reduce the grid graphs, however, it is deemed too computationally expensive for large graphs, so the authors resort to grid partitioning to keep the resulting graphs manageable for sparsification. Our goal is to provide feasible and scalable sparsification of large graphs.

3. Power flow solution

3.1. Problem statement

The subject of the power flow problem is balancing the required load and associated losses with the production capacities of generators in a power grid. The knowledge of one complex characteristic in all nodes of the system, in this case voltage, can be used to reconstruct the complete regime of the system. In other words, the voltage magnitudes and phase angles of each bus constitute the stationary state of the system.

The state of a system of N nodes is defined by N complex equations. The Newton-Raphson method is a well-known approach for solving systems of non-linear equations. Its modus operandi is to approximate a non-linear problem, like a system of N complex power-flow equations, into a linear matrix equation and solve it iteratively.

For massive grids, the corresponding system of linear equations is large. Solving such system is non-trivial, as just computing the inverse of the matrix is challenging, and it has to be done iteratively, which further increases the challenge.

3.2. Solution Approach

The algorithm implemented in our solution is presented in **Fig. 1**. The general idea is to perform fast localized tasks (e.g. initializing the input parameters of the matrix equation and checking the solution) on the driver machine, while distributing compute intensive operations to the worker nodes (e.g. solving the matrix equation).

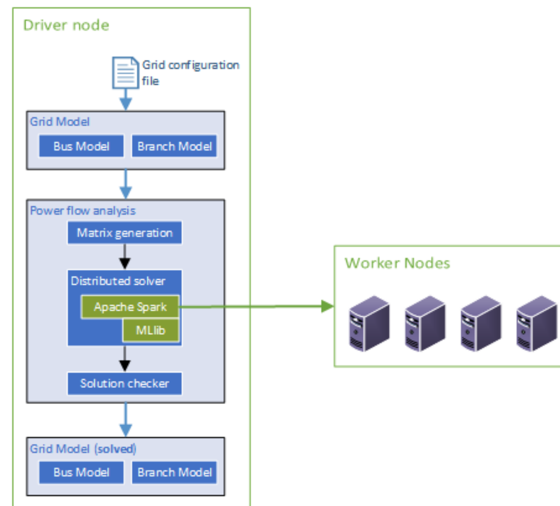


Fig. 1 An overview of the architecture. It shows what parts of the power flow analysis are performed on the driver node and which are distributed across a collection of worker nodes

The input parameter is a GridModel [33], loaded from structured text files that define all known values in a grid. First, it needs to be initialized, i.e. the starting assumptions set and the admittance matrix generated.

Attempts to arrive at a satisfying solution are iteratively made until either the solution is accepted or the maximum number of iterations is reached.

Using Apache Spark, we devised an algorithm that solves the general type of matrix equations ($\mathbf{b} = \mathbf{Ax}$) in a distributed manner. This addresses the hard problem in the power flow calculation [34]: efficiently and repeatedly solving a matrix equation. As a high-level overview of this algorithm, the input parameters are prepared on the driver node and then distributed as Resilient Distributed Datasets (RDDs) [3] across the allocated worker nodes for solving. The result of the operation is passed back to the driver for checking and the setup of the next iteration.

The initial step is parallelizing the input matrix into an RDD. RDDs are special abstractions of collections of objects that represent the basic operating object of all Apache Spark jobs. They are partitioned and distributed across available worker nodes and are fault-tolerant in terms of failure of a job or worker machine. Every RDD has exactly one underlying type which is defined by the collection used for RDD creation.

Spark's MLLib library extends the RDD paradigm by introducing distributed abstractions atop of it. Particularly interesting here are distributed matrix types. For instance, the RowMatrix represents a row-oriented matrix with no meaningful indices, while the IndexedRowMatrix introduces indexed rows. The BlockMatrix views a matrix as a distributed collection of smaller submatrices and a CoordinateMatrix is particularly suitable for sparse matrices, as it is organized as a distributed collection of tuples that define the value of the entry and its coordinates, i.e. the row and column, in the matrix. All of them provide different operations, depending on their logical organization, but converting from one to another must be done carefully, since that may result in

reshuffling of data, arguably the most expensive operation in the Spark environment from a performance perspective.

```
   Algorithm 1: Distributed solving of matrix equation b = A·x
 1.      procedure Solve (Matrix A, Vector x, Vector b)
 2.         rowsRDD ← convert A into RDD of IndexedRows
 3.         indexedRowMatrix ← convert rowsRDD into IndexedRowMatrix
 4.
 5.         sdd ← compute the Singular Value Decomposition of
          indexedRowMatrix
 6.         U ← get the U value from sdd, as IndexedRowMatrix
 7.         S ← get the S value from sdd, as Vector
 8.         V ← get the V value from sdd, as Matrix
 9.
10.         Utrans ←  transpose the value U, as IndexedRowMatrix
11.         UtransB ← Utrans multiplied with Vector b, as
          IndexedRowMatrix
12.         UtransBSinv ← UtransB multiplied with inversed Vector S, as
          RDD of Tuples
13.
14.         UtransBSinvVector ← UtransBSinv collected to local Vector
15.
16.         x ← UtransBSinvVector multiplied with V
```

Fig. 2. Algorithm for distributed solving of matrix equations

The IndexedRowMatrix is particularly interesting for the problem at hand, because it offers two methods of matrix factorization: the QR decomposition and the Singular Value decomposition (SVD). And it offers the additional benefit of indexed rows over the RowMatrix. The only drawback is that, per Spark source code, many operations perform a to-RowMatrix cast first and then issue the operation on the RowMatrix type with optional re-indexing afterwards, which carries a certain performance penalty. The Spark documentation suggests that casting from one distributed type to another may be expensive, however our experience shows that in some scenarios, a cast outperforms an alternative, more complex implementation. This is typically the case when a data shuffle is inevitable, be it performed by a cast or required by a custom implementation.

The choice of a matrix factorization is important for finding the inverse of $\mathbf{A}$ to solve the matrix equation

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \ . \qquad (1)$$

Our choice is in favor of SVD [28]. It produces orthogonal matrices, so their conjugate transpose is at the same time the inverse. Therefore, by performing a SVD, obtaining the inverse of the matrix becomes easier.

Once the matrix equation is solved (**Fig. 2**), a convergence check is performed.

In the case that the current iteration converged, the input GridModel contains the recent changes in terms of bus parameters, i.e. the solved state of the system. The GridModel instance can be used for analysis of the system state or for further simulations.

Finally, in the case when the Newton-Raphson method fails to arrive at an acceptable solution after a given number of iterations, the calculation fails. The predefined maximum number of iteration is also empirically determined. For flat start calculations,

we found that the maximum number of iterations needed is 6 (see experimental results), so 10 iterations present a reasonable maximum iteration threshold.

4. Island detection

4.1. Problem statement

The purpose of contingency analysis is assessing the impact of potential outages on the smart grid. This makes it an important simulation tool.

A contingency analysis is performed in two steps, as shown in **Fig. 3**. First, the connectedness of the network is determined. Second, if any islands were found, their state is assessed by running power flow analyses on each island and checking for irregularities.

The connectedness analysis determines whether all nodes in a network are connected to every other node, at least indirectly. This problem can be reduced to the connected components problem, which is a common challenge in graph theory.

Another method is the full matrix analysis [29], which may be considered the "classic approach". The idea is to generate a Boolean connectedness matrix whose element with the indices *i* and *j* is set to one, if nodes *i* and *j* are connected, and zero otherwise. The matrix is symmetrical. The elements on the main diagonal are always set to one.

```
Algorithm 2: Contingency analysis
1.     procedure PerformContingencyAnalysis (GridModel gridModel,
     Branch from, Branch to)
2.         baseGridModel ← PowerFlowCalculation(gridModel)
3.         gridModelWithOutage ← CreateOutage(gridModel, from, to)
4.
5.         islandGridModels ←
       PerformConnectednessAnalysis(gridModelWithOutage)
6.
7.         foreach island in islandGridModels do
8.             PowerFlowCalculation(island)
9.             Identify irregularities in grid parameters
```

Fig. 3. Contingency analysis algorithm

Binary multiplying the connectedness matrix with itself yields a connectedness matrix of the second level. If an element with the indices *i* and *j* was zero in the first level matrix, but changed to one in the second level matrix, that means that the nodes *i* and *j* are indirectly connected with one other node in-between. Further binary multiplication with the resulting matrix yields connectedness matrices of higher levels, each identifying deeper connections between disconnected nodes. Consequently, in a system of N nodes, the algorithm stops after (N-2) iterations. If the initial matrix, raised to the (N-2) power through binary multiplication, still contains elements equal to zero, these nodes have a distance larger than N elements between each other. Therefore, that set of zero-valued elements indicates components that are not connected to the main body of the network.

A fully connected network would result in a matrix where all the elements are equal to one.

One stopping criterium is checking whether the (N-2)-th power of the initial connectedness matrix contains any zero-valued elements. However, it can be relaxed. The network is fully connected, if any binary multiplication arrives at an all-ones matrix. The algorithm can then be halted, since further multiplications will not change the outcome. The more general halting condition is therefore that two subsequent multiplications did not change the resulting matrix. That means there can be no more connections, since no new links between any nodes were discovered.

In the final matrix, islands are identified by analyzing its rows or columns for linear dependence. Each linearly independent row or column represents an island. The nodes constituting it are defined by the indices of all the elements equal to one within that row or column.

4.2. Solution Approach

It is our goal to distribute the compute intensive work as much as possible. For that purpose, we implemented two approaches. The first uses the Apache Spark GraphX [25] library to analyze the network with graph analysis and the second uses binary matrix multiplication to isolate the potential islands. A high-level overview of the solution is shown in **Fig. 4**.

As above, the input is the GridModel.

We need to determine a base state of the system before simulating any outages, by performing a power flow calculation with the unchanged GridModel. The base state serves to assess the deviations introduced by outages once the procedure completes.

Performing a connectedness analysis, whose output is a list of potential islands, is where the two mentioned implementations differ. Both approaches share the same interface.

We shall first discuss the graph approach, as it is more straightforward and then move on to the matrix binary multiplication algorithm.

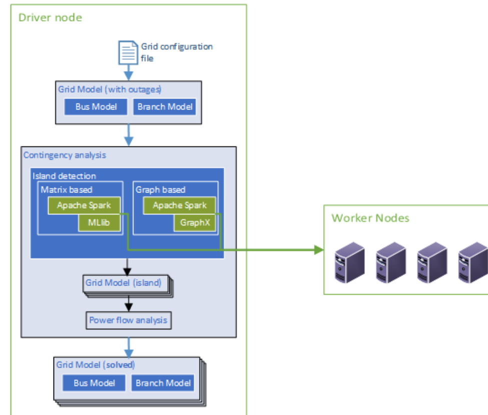


Fig. 4. An overview of the architecture. It shows what parts of the contingency analysis are performed on the driver node and which are distributed across a collection of worker nodes

Representing a smart grid network with a graph comes naturally. A vertex collection is created from all buses in the bus model. Edges are similarly created from the branch model. The GraphX graph object is created using those two RDD collections. The graph object supports a connectedComponents operation, which returns another graph object. Practically, transforming the vertex RDD into a list of islands and returning this list to the driver, solves the problem.

The matrix binary multiplication approach is still the de facto standard method, albeit with significant improvements developed over the years, for many industrial analysis systems. Further, distributed matrix operations, including multiplication, are the domain of Spark's MLlib [30]. So, the reasons of legacy consideration and challenge to implement a distributed binary matrix multiplication in Spark prompted us to develop the second island detection approach.

Our distributed binary matrix multiplication operates in three phases:

1. Generate the initial connectivity matrix based on the state of the system
2. Find the stable island matrix
3. Identify the islands from the island matrix

The connectivity matrix is generated from the branch model. For its generation it is convenient to use a CoordinateMatrix type, because the underlying MatrixEntry, which consists of the two coordinates and the value, best fulfills the need to set a number of elements individually. Further, as the connectivity matrix represents the physical connections between the nodes in the network, it is very sparse, especially for large systems, so relatively few MatrixEntries are required. Yet, because the BlockMatrix is the only matrix type that supports multiplication with another distributed matrix, which is the focus of this algorithm, the generated CoordinateMatrix is cast to a BlockMatrix at the end of this phase. This cast does not induce any significant performance penalty, as opposed to directly creating a BlockMatrix, because of the sparsity of the matrix and the overhead the block-based approach would require during the initialization.

The next phase determines the island matrix. Taking the BlockMatrix from the previous phase, it is repeatedly multiplied with itself. The maximum number of multiplications depends on the number of buses in the system and is equal to

$\lceil \log_2 (n_{buses} - 2) \rceil$. The goal is to raise the connectivity matrix to the $(n_{buses} - 2)$ power and this is a much more efficient way. After each multiplication, all values greater than zero are set to one. The procedure halts when the maximum number of iterations is reached or when the matrix is completely filled with ones. A real-world, connected network usually yields the all-one matrix after few iterations.

The final phase evaluates the island matrix for occurring islands. To keep the result consistent with the graph approach, a connected network results in a single list of all nodes, with the consideration that that network contains the designated number of outages.

Regardless of chosen approach, the obtained list of islands is evaluated. From each list a new GridModel is created that matches the subnetwork of the island.

5. Spectral graph sparsification

5.1. Problem statement

The most interesting fact about spectral sparsification is that it is possible to sparsify every graph in this way. In a narrower sense, spectral sparsification implies that a graph $G = (V, E, w)$ can be approximated by a sparse subgraph that retains the same Laplacian quadratic form as the original graph [31]. The Laplacian quadratic form is given by

$$x^T L_G x = \sum_{(u,v)\in E} w_{(u,v)} \big(x(u) - x(v)\big)^2, \quad (14)$$

where x is a real vector of V elements, and L_G is the Laplacian matrix of G [12][17]. Strictly speaking, Spielman and Teng [17] consider $\tilde{G}$ to be a σ-sparsification of G if the following relation holds for all x:

$$\frac{1}{\sigma} x^T L_{\tilde{G}} x \le x^T L_G x \le \sigma x^T L_{\tilde{G}} x. \quad (15)$$

Basically, finding $\tilde{G}$ for a given graph G is the aim of our solution and its core method *sparsify*.

5.2. Solution Approach

Overview

Here we are extending the existing open source spectral graph library [23][16], that builds upon the theoretical algorithms and examples of Spielman et al [17]-[20][31]. It already has some basic methods, some of which are used for the subsequent implementation.

We extend the public API with the following methods: *volume*, *conductance*, *sum*, and *sparsify*. The starting point for the considerations is always graph G, the GraphX graph object the class is initialized with and which serves as the primary input. When we discuss edges, vertices, subgraphs, etc., it is done with regard to G unless otherwise noted. Further, the cornerstone of the approach is viewing the problem from the set perspective. By focusing on that aspect, the set of vertices, edges, weights, i.e. the usual ways to define a graph, become natural subjects of GraphX and the underlying Spark RDD [1] paradigm. Thus, it is easy to scale a large graph to a distributed environment and making computationally expensive operations feasible on a large dataset.

Volume

Volume calculates the volume of a subset of vertices. It is a sum of the degrees of each vertex in the set. The degree of a vertex is equal to the number of its incident edges. Here, we distinguish two cases. The volume is constant, if the subset is in fact the whole set of vertices of G, and is equal to twice the number of edges in G. In the case of a true subset, we first calculate the degrees of all vertices and create a set of tuples that matches each vertex' unique ID with its degree. The resulting set is joined with the subset of vertices by vertex ID, leaving only the vertices that were part of the subset mapped together with their respective degrees. This set is reduced to a sum of the degrees within producing the required volume value.

Conductance

The conductance of a graph, also called Cheeger's constant, is formally related to the convergence of a random walk on the graph to a uniform distribution. The name derives from the similarity to the significance of random walks in electrical networks. Here, conductance is used primarily for graph partitioning in the sense of evaluating the quality of a local cluster. A cluster is considered of high quality, if it is extensively interconnected within itself, but rather sparsely with the rest of the graph. In other words, conductance is the ratio of the number of edges connecting the cluster with the rest of the graph and the number of edges within [18].

In this solution, the conductance is calculated with regard to a given vertex set, i.e. a cluster, that is part of a given subgraph, which in the general case can also be the whole graph G. First, we calculate the number of edges crossing out of the set, i.e. we identify the edges whose one vertex is inside the set, while the other reaches outside. Next, the volumes of both the given set of vertices and the volume of the remaining set of vertices in the whole graph G are calculated. At this point it is important to emphasize, that when measuring and volumes of vertices in the vertex-induced subgraphs, we will continue to measure the volume according to the degrees of vertices in the original graph G. As mentioned above, the resulting conductance is the ratio of the obtained inter-cluster edge count and the smaller of the two volumes.

Sum

The concept of adding two or more graphs together might seem counterintuitive. However, as we will see later on, an important step in graph sparsification is the so-called partitioning into certain subgraphs, whose sum results in a single sparsified graph.

In the spectral sense, a sum of two graphs produces a graph whose Laplacian matrix is equal to the sum of their Laplacian matrices. In practical terms, this means that every edge in the resulting graph is equal to the sum of the corresponding edges in the two constituent graphs. In the case that the vertex sets of the corresponding graphs are disjoint, the sum is a simple union of graphs.

The resulting graph is defined by its vertex and edge sets. Obtaining the vertex set is trivial, it is only a simple union of the vertex sets of the two addend graphs. In the case that an edge is part of both addends, the corresponding edge in the resulting graph will have a weight equal to the sum of their weights. If an edge is only present in one of the addends, the same principle applies, while the non-existing edge will be treated to have weight zero. One problem that arises is the uniform designation of edges. In the general case, an edge is defined by its source vertex, destination vertex, and weight. Given that we consider here undirected graphs, there is no distinction between edges that have their (same) vertices swapped, i.e. for summation purposes, such edges are considered the same and should be added accordingly. However, there is no guarantee nor binding rule that the edges in the addends are not swapped. To alleviate this case, we manipulate both edge sets by assuring an ascending order of vertex IDs in the tuple that defines the edge, i.e. $(a, b) => (b, a), if\ a > b$. This provides a kind of unique key so that the corresponding edges, that generally can have different weights, can be joined together by the vertices they connect, which results in set that uniquely maps the incident vertices to the weights that the corresponding edge has in both addend edge sets. A full outer join guarantees that even when an edge is not present in the other graph, it will still be present in the joined set with a special designation (concretely, the type None) depicting the “missing” weight. The resulting edge set is obtained by conditionally summing up the weights and keeping the vertex IDs. Thus completing the other requirement for the sum of two graphs.

Sparsify

Sparsify is a complex method at the core of the sparsification process. Broadly speaking, it consists of two major steps, that are distinctive, yet conjoined through common weight adjustments. First, we partition the graph and sample the resulting subgraphs’ edge sets. Then, the resulting graphs are contracted together into a single sparsified graph.

In order to support graphs with arbitrary weights, sparsify is limited to graphs that have fractal weights that are greater than zero and at most equal to one. This restriction can be easily overcome by simply scaling all weights down before sparsifying and scaling them up afterwards. This is another task that is rather trivially fulfilled using Spark RDD operations, even for very large graphs. However, the problem here is not the possibility and cost of scaling, but the fact that the weights can be truly arbitrary, i.e. a large number of digits after the decimal point, and increase the complexity of the calculation and can lead to the inability to construct a sparsified graph. Therefore, the

weights are scaled down to a certain number of bits after the decimal point before proceeding. Depending on this number of bits, the same number of subgraphs is created based on the binary representation of the individual adjusted edge weights. Each of those subgraphs is then subjected to the following operations.

Before proceeding to the partitioning and sampling of the graph, there is an issue worth mentioning. There is a reported, and as of yet not completely resolved, issue [32] that under certain circumstances the execution of *connectedComponents* gets stuck in an endless loop. We did occasionally observe such behavior when working with massive graphs. Given that the issue seems also related to the workload of the environment and other factors, the simplest workaround is to restart the calculation.

When partitioning, the goal is to isolate a portion of the graph with a specified target conductance [18]. If the obtained partition is large, i.e. has a large conductance, both the partition and its remaining complement are recursively cut further, until the target conductance is reached. If a partition fulfills the target, further cutting is applied to its complement, until it too reaches the target. The result of the partitioning is a collection of subgraphs, that are induced by the obtained sufficiently small cuts. Each of the subgraphs is sampled, which is a random procedure, where the subgraph's vertex set remains unchanged while the edges are scaled and reduced based on a probability distribution. Formally [18], this step creates a $(1 + \epsilon)$-approximation of the subgraph, where the ϵ is a rational parameter. This subgraph collection is, once the processing is completed, summed up back into a single graph. Such resulting graphs can already be considered sparsified to a degree.

At this moment, the initial graph has been decomposed into edge induced subgraphs, following the realignment of the edges' weights. Each of these subgraphs is individually partitioned and sampled into a sparsified version of itself, as we saw above. We further sparsify the current subgraph's edges by identifying those that contribute the least to the overall conductance of the subgraph. All that remains is to sum those modified subgraphs into a single sparsified graph that is returned as the result of the operation.

At a high level, the algorithm repeatedly breaks the graph into ever smaller, yet mostly overlapping, subgraphs and attempts to reduce the number of edges at each step. We observe another similar pattern at each of the sparsification steps. Each of the currently relevant subgraphs is broken up into a collection of subgraphs, their edges processed in some way, and reduced to a single graph by adding them up together (the operation Sum from above). Nothing is lost due to those repeated breakdowns, even as the subgraphs are usually overlapping, because the addition of graphs preserves the vertices and only manipulates the weights of edges that exist in any of the subgraphs. The abovementioned operations Conductance and Volume are used as limits while partitioning graphs.

6. Experimental results

6.1. Environment

For the evaluation computing infrastructure, we've chosen Amazon cloud computing platform, Amazon Web Services (AWS). Testing was conducted on configurations of Amazon Elastic Compute Cloud (EC2) instances. The performance results are limited to computing optimized (c types) and general-purpose GPU compute instances (p types), where applicable (see **Table 1**). Both are suitable for computation centered parallel tasks, which conceptually fits the needs of this paper. However, although significantly more powerful, the GPU-based instances proved to offer little performance gain, as seen below. The reason is that at the moment of testing, Spark didn't support the utilization of CUDA cores. The execution practically ignored the available GPU cores and focused solely on the CPU cores.

The experiments are executed using the Amazon Elastic MapReduce (EMR) framework. It offers an abstraction over the "vanilla" EC2 instances, which allows Spark and any related services (e.g. Ganglia, S3 support, etc.) to be automatically deployed and accessed when starting up a cluster.

Our experiments typically include two configurations for each instance type, in order to gauge the scaling-out of the solution. One that has five workers and one of ten, while both have one driver machine.

Table 1. EC2 instances used in the experiments and their technical specification

Instance type	CPU cores	GPU cores	GPU RAM	RAM
c3.4xlarge	16	n/a	n/a	30 Gb
c3.8xlarge	32	n/a	n/a	60 Gb
p2.8xlarge	32	8	96 Gb	488 Gb
p2.16xlarge	64	16	192 Gb	732 Gb

6.2. Test cases

For evaluating the smart grid analysis part of the solution, we used a collection of test cases which encompass experimental and special case networks, as well as real-world installations. A useful source of preselected grid examples is also the case repository that comes with MatPower [11]. We have conducted experiments on an array of 21 cases, with bus size ranging from 4 to 9241.

These networks proved unsuitable for proper graph sparsification demonstration, because they are not large enough. We chose two real world graph datasets for experimental evaluation. The data is publicly available [24] and represents weighted graphs. The first dataset is called "bio-mouse-gene" and represents a mouse gene regulatory network derived from analyzing gene expression profiles. It consists of 45101 vertices and 14506196 edges. The other dataset is called "bio-human-gene2" and similarly represents a gene regulatory network, but this time for humans. It consists of

14340 vertices and 9041364 edges. These graphs were chosen, because they have both semantic significance, i.e. are not artificially generated, and similarity, i.e. they represent genetic networks. Further, these are graphs that have a relatively high edge count, different densities, weight distributions, and topographical layouts.

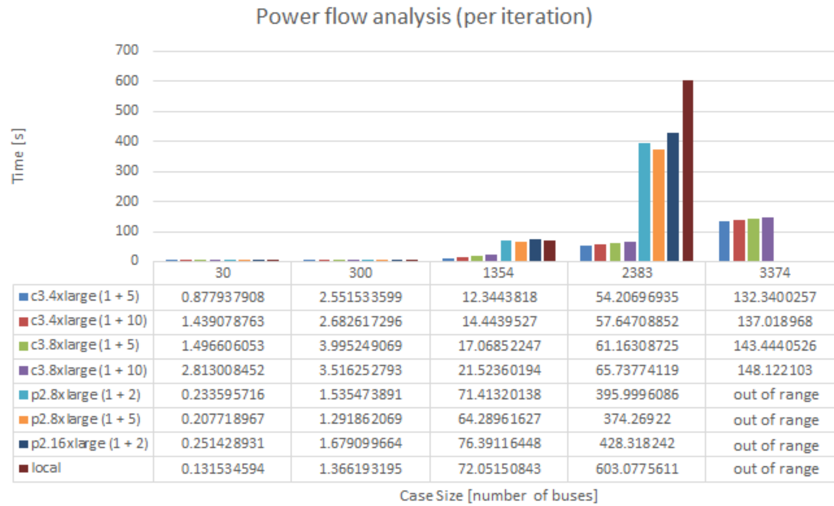
It is important to point out that gene networks hold no special significance for the solution's approach or performance. For example, power networks and smart grids are also excellent inputs, especially due to the importance of weights (which are usually line admittances) in such graphs. However, such networks that are publicly available, tend to be rather small for demonstration purposes. We want to emphasize here the ability of our solution to operate in line with performance expectations of Spark, which excels in large datasets.

6.3. Results and discussions

Here, we show the experimental results achieved in various environments and parameters. The graphical representations follow a pattern in order to facilitate understanding: The vertical axis shows the subject being measured (e.g. execution time or reduced edge count), the horizontal axis shows the parameters used in each experiment, while the table that follows shows the exact values as opposed to graph lines. The first column there indicates the configuration on which the value was obtained (e.g. the number of worker nodes and the EC2 instance types).

Power flow solution

The execution results of a series of experiments under which the power flow calculation was tested, is shown in **Fig. 5**. The running times are scaled to the duration of one iteration of the Newton-Raphson method, as it takes a variable number of iterations to complete different cases (**Table 2**). We found that on average less than 2% of the total duration of a calculation is spent on the driver node preparing the current iteration (e.g. generating the Jacobian and $\Delta\mathbf{S}$ matrices) and evaluating the obtained results (checking the validity of the solution).



	30	300	1354	2383	3374
c3.4xlarge (1 + 5)	0.877937908	2.551533599	12.3443818	54.20696935	132.3400257
c3.4xlarge (1 + 10)	1.439078763	2.682617296	14.4439527	57.64708852	137.018968
c3.8xlarge (1 + 5)	1.496606053	3.995249069	17.06852247	61.16308725	143.4440526
c3.8xlarge (1 + 10)	2.813008452	3.516252793	21.52360194	65.73774119	148.122103
p2.8xlarge (1 + 2)	0.233595716	1.535473891	71.41320138	395.9996086	out of range
p2.8xlarge (1 + 5)	0.207718967	1.291862069	64.28961627	374.26922	out of range
p2.16xlarge (1 + 2)	0.251428931	1.679099664	76.39116448	428.318242	out of range
local	0.131534594	1.366193195	72.05150843	603.0775611	out of range

Fig. 5. The performance results of the power flow analysis, scaled to per-iteration values

Table 2. Number of iterations per power flow calculation

Buses	30	300	1354	2383	3374
Iterations	3	5	5	6	6

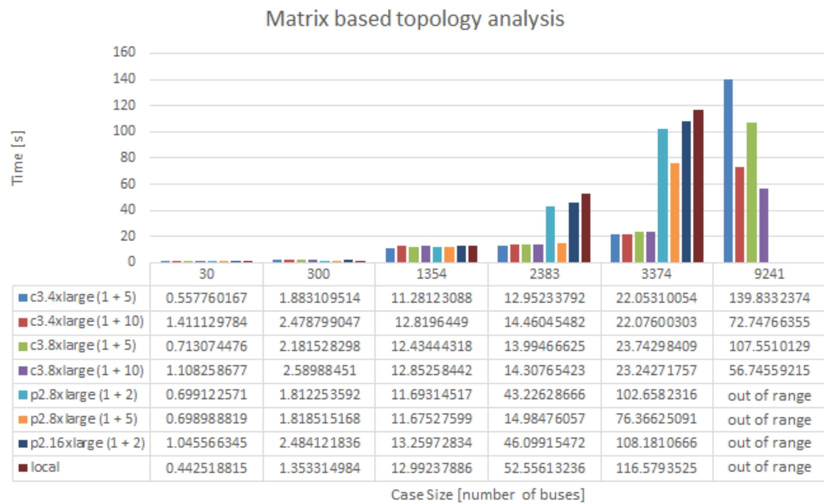
Measured times indicate a relative equality between distributed and local executions for lower bus-counts. We even observe that the local performance is better than the distributed equivalent. This is expected behavior, since the datasets are small enough that the overhead of partitioning them into many more partitions, sending them to the workers, and collecting them back, exceeds the computing effort itself. Further, it also shows that reasonably small cases are practically computable on single machines, which is why commercial power analytics software systems tend to break the network into smaller chunks before performing a power-flow calculation. However, the differences rapidly escalate with the growth of the network. For the grid of 3374 buses it became unfeasible to chart the result of the local execution, as it was in the domain of several hours. This is where the advantages of a distributed implementation truly outperform the local equivalent. Unfortunately, Spark's ability to perform a SVD is limited to matrices of at most 17515 columns. In the source code of the RowMatrix class, this is explained with the rationale that the matrix dimension "exceeds the breeze svd capability", so it is the issue of the Breeze [27] package, a numerical processing library for Scala that Spark uses. Thus, we were unable to perform power-flow analyses for the 9241-bus and larger grids. This also explains the slight performance degradation when scaling out and upgrading the c3 instances. Namely, the idea is to fully utilize the available computation power as much as possible. The 3374-bus test case had an average CPU utilization of 80% percent, when performed on a cluster of five workers, while the same case used no more than 30% percent on a cluster of ten instances. Once the SVD size limitation is

alleviated, the full performance impact can be gauged with larger networks, which would make full use of the available cluster resources.

Experiments on GPU instances were conducted using Databricks' Spark GPU. Due to the inhibition of the Spark functionality, they've shown similar performance as the local machine. The minor performance benefit is due to more advanced CPU and memory configuration.

Island detection solution

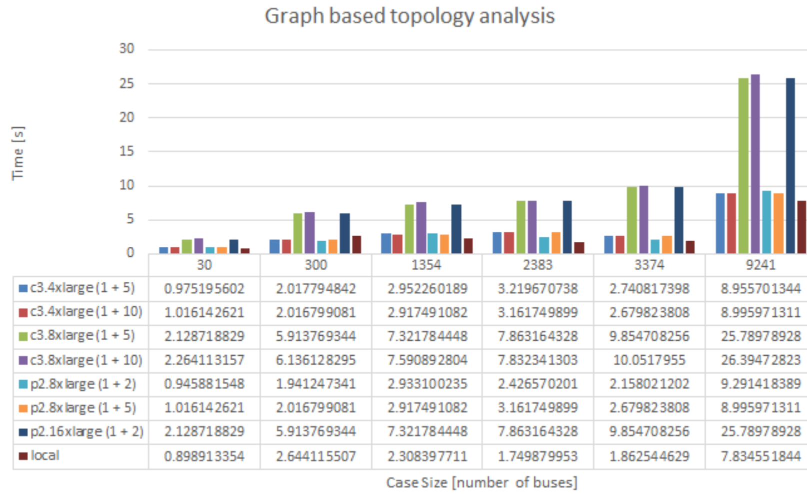
Measured times indicate a relative equality between distributed and local executions for lower bus-counts. We even observe that the local performance is better than the distributed equivalent. This is expected behavior, since the datasets are small enough that the overhead of partitioning them into many more partitions, sending them to the workers, and collecting them back, exceeds the computing effort itself. Further, it also shows that reasonably small cases are practically computable on single machines. However, the differences rapidly escalate with the growth of the network and the advantages of a distributed implementation truly outperform the local equivalent.



	30	300	1354	2383	3374	9241
c3.4xlarge (1 + 5)	0.557760167	1.883109514	11.28123088	12.95233792	22.05310054	139.8332374
c3.4xlarge (1 + 10)	1.411129784	2.478799047	12.8196449	14.46045482	22.07600303	72.74766355
c3.8xlarge (1 + 5)	0.713074476	2.181528298	12.43444318	13.99466625	23.74298409	107.5510129
c3.8xlarge (1 + 10)	1.108258677	2.58988451	12.85258442	14.30765423	23.24271757	56.74559215
p2.8xlarge (1 + 2)	0.699122571	1.812253592	11.69314517	43.22628666	102.6582316	out of range
p2.8xlarge (1 + 5)	0.698988819	1.818515168	11.67527599	14.98476057	76.36625091	out of range
p2.16xlarge (1 + 2)	1.045566345	2.484121836	13.25972834	46.09915472	108.1810666	out of range
local	0.442518815	1.353314984	12.99237886	52.55613236	116.5793525	out of range

Fig. 6. The performance results of the matrix based topology analysis

The performance chart of matrix based topology analysis is shown in **Fig. 6**. We observe performance improvement when increasing the number of workers and using better EC2 instances. The importance of scale-out is also evident, ten c3.4xlarge instances outperform five c3.8xlarge workers by about 30%. While in the extreme case, ten c3.8xlarge instances are close to three times faster than five c3.4xlarge workers. If we additionally consider that smaller networks show similar performance differences between c3 configurations, the results of the matrix topology analysis support the claim, that larger networks would make full use of the available cluster resources.



	30	300	1354	2383	3374	9241
c3.4xlarge (1 + 5)	0.975195602	2.017794842	2.952260189	3.219670738	2.740817398	8.955701344
c3.4xlarge (1 + 10)	1.016142621	2.016799081	2.917491082	3.161749899	2.679823808	8.995971311
c3.8xlarge (1 + 5)	2.128718829	5.913769344	7.321784448	7.863164328	9.854708256	25.78978928
c3.8xlarge (1 + 10)	2.264113157	6.136128295	7.590892804	7.832341303	10.0517955	26.39472823
p2.8xlarge (1 + 2)	0.945881548	1.941247341	2.933100235	2.426570201	2.158021202	9.291418389
p2.8xlarge (1 + 5)	1.016142621	2.016799081	2.917491082	3.161749899	2.679823808	8.995971311
p2.16xlarge (1 + 2)	2.128718829	5.913769344	7.321784448	7.863164328	9.854708256	25.78978928
local	0.898913354	2.644115507	2.308397711	1.749879953	1.862544629	7.834551844

Fig. 7. The performance results of the graph based topology analysis

Experimental results of the graph based topology analysis, **Fig. 7**, show that it clearly outperforms its matrix equivalent. A significant conclusion is that for such small graphs, there is no real need to distribute the work. The increased computing power of a cluster is largely unused and overshadowed by the overhead costs.

Spectral graph sparsification

The *sparsify* method accepts two parameters, ϵ and p. According to the sparsification theorem by Batson et al. [31], the parameters are bounded as follows: $\epsilon \in \left(\frac{1}{n}, \frac{1}{3}\right)$ and $p \in \left(0, \frac{1}{2}\right)$, where n is the number of vertices. As we had no general guideline how to choose values in order to produce optimal results, we selected uniformly a few values, shown in **Table 3**, to show how the algorithm behaves at various parts of the bounded spectrum.

Table 3 Parameters used as sparsify input

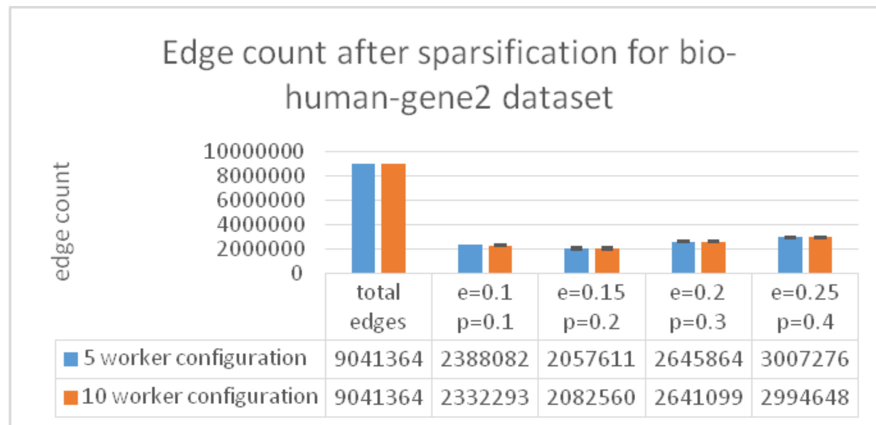
ϵ	0.1	0.15	0.2	0.25
p	0.1	0.2	0.3	0.4

The experiment procedure included parsing the graph data into a GraphX graph object and sparsifying it. We analyzed two outputs: the time it took to finish the sparsification (excluding the preparation time) and the degree of the sparsification, i.e. how many edges were left afterwards. As said above, we had two setups, with five and ten worker nodes. On each, all listed parameter pairs were used and iterated a number of times in order to obtain mean values for both outputs.

	e=0.1 p=0.1	e=0.15 p=0.2	e=0.2 p=0.3	e=0.25 p=0.4
bio-human-gene2 (5 workers)	5914.869081	5380.573035	6097.81851	6209.277437
bio-human-gene2 (10 workers)	4153.301844	3732.046778	4227.041625	4313.345095
bio-mouse-gene (5 workers)	9990.18734	9575.352944	10244.72488	10628.05124
bio-mouse-gene (10 workers)	7111.963286	6458.472137	7384.569416	7515.986213

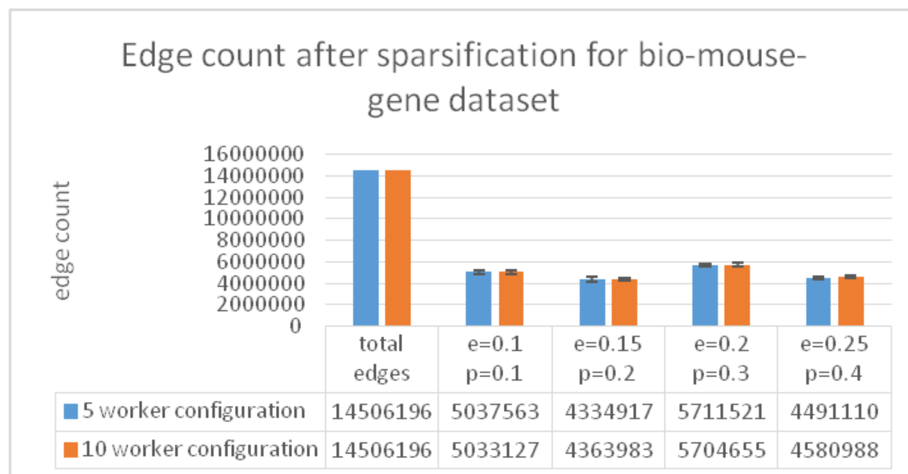
Fig. 8. The times it took to sparsify both datasets in both environments consisting of one master node and five and ten worker nodes

The execution time results are shown in **Fig. 8**. We observe a performance gain of approximately 30% when scaling up to ten workers. Given that we kept the cluster and Spark configuration as uniform as possible during the experiments, there is still room for fine tuning, which could yield better results. The observed deviation is explained by two dominant factors. First, the random nature of the sampling procedure, and, second, the distributed cloud environment, which cannot guarantee the same conditions for every execution. It is interesting to note that the performance gained through parameter modification is rather consistent across the iterations, which indicates that the choice of parameter values provides another tuning opportunity, independent from the random process.



	total edges	e=0.1 p=0.1	e=0.15 p=0.2	e=0.2 p=0.3	e=0.25 p=0.4
5 worker configuration	9041364	2388082	2057611	2645864	3007276
10 worker configuration	9041364	2332293	2082560	2641099	2994648

Fig. 9. The degree of sparsification, i.e. the remaining edge counts after the calculation, for the bio-human-gene2 dataset. The results are shown for some parameter combinations and compared to the edge count of the dataset before sparsification



	total edges	e=0.1 p=0.1	e=0.15 p=0.2	e=0.2 p=0.3	e=0.25 p=0.4
5 worker configuration	14506196	5037563	4334917	5711521	4491110
10 worker configuration	14506196	5033127	4363983	5704655	4580988

Fig. 10. The degree of sparsification, i.e. the remaining edge counts after the calculation, for the bio-mouse-gene dataset. The results are shown for some parameter combinations and compared to the edge count of the dataset before sparsification

Fig. 9 and **Fig. 10** show the achieved sparsification for each dataset. The graphs compare the edge count of the initial dataset with the edges obtained after the sparsify procedure with the same parameter variations and environments as before. First of all, note the magnitude of sparsification. In some instances, the resulting edge set is less than 30% of the original one. That means that we can get a graph, spectrally similar to the original one, with just a third of the starting edges. Again we observe a slight deviation in the resulting edge counts. This means that, all the things being the same, we can expect to get a different number of edges across multiple runs. Although that may seem

surprising, it is a consequence of the random process. Also, one should keep in mind, that the variance is negligible compared to the edges count and that the aim of the sparsification is to get an approximation of the initial graph. The choice of parameter values has a more profound impact here than on the previous time analysis. We can observe somewhat of a trend, different for both datasets, however, it indicates the existence of a minimum configuration, which would result in the most sparsification.

7. Conclusion

Our primary goal in this paper was to reflect on complex algorithms and adapt them to the requirements of the Big Data era. At the core lie mathematical problems that are generic enough to be applicable to broader spectrum of applications. We achieved this by contributing open source solutions for a few chosen problems and showing their performance under load.

Our three projects demonstrate how complex and computationally demanding solutions are applied to large datasets in a distributed and scalable environment. The operations themselves are illustrative to the broad spectrum of algorithmic approaches that can be optimized in this manner.

Underlying the power flow problem is the Newton-Raphson iterative method. The system size directly affects the number of required equations and thus the size of the problem.

The island detection problem compares two approaches. Iterative binary matrix multiplication addresses another common complex algebraic operation. The connectedness analysis is also an important tool in graph manipulations.

Spectral graph analysis is a relatively specific concept that demonstrated useful applications. It is still a complex and demanding procedure, however, we showed it could be effectively parallelized and thus made applicable to even the most complex of graphs.

The experimental results indicate that the solutions perform well on large datasets and that they easily scale.

There is room for performance improvement as well as expanding the existing toolkit with more solutions. Further research can also be directed towards combining the presented solutions to other derivative applications.

Reference

1. Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.
2. Li, Xueqi, et al. "Accelerating large-scale genomic analysis with Spark." Bioinformatics and Biomedicine (BIBM), 2016 IEEE International Conference on. IEEE, 2016
3. Ji, Hao, et al. "An Apache Spark implementation of block power method for computing dominant eigenvalues and eigenvectors of large-scale matrices." Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable

Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on. IEEE, 2016.
4. Van Ness, James E. "Iteration methods for digital load flow studies." Transactions of the American Institute of Electrical Engineers. Part III: Power Apparatus and Systems 78.3 (1959): 583-586.
5. Tinney, William F., and Clifford E. Hart. "Power flow solution by Newton's method." IEEE Transactions on Power Apparatus and systems 11 (1967): 1449-1460.
6. Trias, Antonio. "The holomorphic embedding load flow method." Power and Energy Society General Meeting, 2012 IEEE. IEEE, 2012.
7. Goderya, F., A. A. Metwally, and O. Mansour. "Fast detection and identification of islands in power networks." IEEE transactions on power apparatus and systems 1 (1980): 217-221.
8. Montagna, M., and G. P. Granelli. "Detection of Jacobian singularity and network islanding in power flow computations." IEE Proceedings-Generation, Transmission and Distribution 142.6 (1995): 589-594.
9. Guler, Teoman, and George Gross. "Detection of island formation and identification of causal factors under multiple line outages." IEEE Transactions on Power Systems 22.2 (2007): 505-513.
10. Stott, Brian, Ongun Alsac, and Alcir J. Monticelli. "Security analysis and optimization." Proceedings of the IEEE 75.12 (1987): 1623-1644.
11. Zimmerman, Ray Daniel, Carlos Edmundo Murillo-Sánchez, and Robert John Thomas. "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education." IEEE Transactions on power systems 26.1 (2011): 12-19
12. Beerten, Jef, and Ronnie Belmans. "Development of an open source power flow software for high voltage direct current grids and hybrid AC/DC systems: MATACDC." IET Generation, Transmission & Distribution 9.10 (2015): 966-974.
13. Li, Hongyan, Junjie Sun, and Leigh Tesfatsion. "Dynamic LMP response under alternative price-cap and price-sensitive demand scenarios." Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century, 2008 IEEE. IEEE, 2008.
14. Zhou, Michael, and Shizhao Zhou. "Internet, open-source and power system simulation." Power Engineering Society General Meeting, 2007. IEEE. IEEE, 2007.
15. https://bitbucket.org/suticd/sparkpowertools/src/master/
16. https://bitbucket.org/suticd/spectralgraphanalysistool/src/master/
17. Spielman, Daniel A., and Shang-Hua Teng. "Spectral sparsification of graphs." SIAM Journal on Computing 40.4 (2011): 981-1025.
18. Spielman, Daniel A., and Shang-Hua Teng. "A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning." SIAM Journal on Computing42.1 (2013): 1-26.
19. Spielman, Daniel A., and Shang-Hua Teng. "Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems." SIAM Journal on Matrix Analysis and Applications 35.3 (2014): 835-885.
20. Spielman, Daniel A., and Nikhil Srivastava. "Graph sparsification by effective resistances." SIAM Journal on Computing 40.6 (2011): 1913-1926.
21. Koutis, Ioannis, and Shen Chen Xu. "Simple parallel and distributed algorithms for spectral graph sparsification." ACM Transactions on Parallel Computing (TOPC) 3.2 (2016): 14.
22. Sun, He, and Luca Zanetti. "Distributed graph clustering and sparsification." ACM Transactions on Parallel Computing (TOPC) 6.3 (2019): 17.
23. Šutić, Davor, and Ervin Varga. "Spectral Graph Analysis with Apache Spark." Proceedings of the 2018 International Conference on Mathematics and Statistics. ACM, 2018.
24. Rossi, Ryan, and Nesreen Ahmed. "The network data repository with interactive graph analytics and visualization." Twenty-Ninth AAAI Conference on Artificial Intelligence. 2015.

25. Xin, Reynold S., et al. "Graphx: A resilient distributed graph system on spark." First International Workshop on Graph Data Management Experiences and Systems. ACM, 2013.
26. Zhao, Xueqian, Zhuo Feng, and Cheng Zhuo. "An efficient spectral graph sparsification approach to scalable reduction of large flip-chip power grids." Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design. IEEE Press, 2014.
27. Jancauskas, Vytautas. “Scientific Computing with Scala.” Packt Publishing Ltd, 2016
28. Hong, Yoo Pyo, and C-T. Pan. "Rank-revealing QR factorizations and the singular value decomposition." Mathematics of Computation 58.197 (1992): 213-232.
29. Goderya, F., A. A. Metwally, and O. Mansour. "Fast detection and identification of islands in power networks." IEEE transactions on power apparatus and systems 1 (1980): 217-221.
30. Bosagh Zadeh, Reza, et al. "Matrix computations and optimization in apache spark." Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2016.
31. Batson, Joshua, et al. "Spectral sparsification of graphs: theory and algorithms." Communications of the ACM 56.8 (2013): 87-94.
32. https://issues.apache.org/jira/browse/SPARK-10335
33. Šutić, Davor, and Ervin Varga. " Appendix - Grid model", https://bitbucket.org/suticd/sparkpowercalculations/src/master/Documentation/Appendix%20-%20Grid%20Model.pdf
34. Šutić, Davor, and Ervin Varga. " Appendix - Power flow problem formulation", https://bitbucket.org/suticd/sparkpowercalculations/src/master/Documentation/Appendix%20-%20Power%20flow%20problem%20formulation.pdf
35. Perraudin, Nathanaël, Johan Paratte, David Shuman, Lionel Martin, Vassilis Kalofolias, Pierre Vandergheynst, and David K. Hammond. "GSPBOX: A toolbox for signal processing on graphs." arXiv preprint arXiv:1408.5781 (2014).

Davor Šutić was born in 1987. He holds the BSc degree in Computer Science from the School of Electrical Engineering, Belgrade, Serbia and the MSc degree from the Faculty of Technical Sciences, Novi Sad, Serbia. Currently, he is a PhD candidate at the same school.

Ervin Varga was born in Kikinda, Serbia on 19.11.1970. He graduated in 1994 and earned his BSc title in electrical engineering at the University of Novi Sad, Faculty of Technical Sciences Novi Sad, Serbia. In 1999 he finalized his master studies and earned the MSc title in computer science at the same university. Ervin defended his PhD thesis in 2007 and earned the PhD title in electrical engineering (his thesis was an application of software engineering and computer science in the domain of electrical power systems) at the same university. Ervin is a Senior Member of the IEEE.

Received: May 31, 2020; Accepted: July 15, 2021.