Spatiotemporal analysis of the resilience of the Lebanese power grid using Apache Spark

Fatima Abu Salem[1,*], Mohamad Jaber[1], Omar Mehio[1], Chadi Abdallah[2] and Sara Najem[2]
[1] *Computer Science Department, American University of Beirut, Beirut, Lebanon and*
[2] *National Center for Remote Sensing, National Council for Scientific Research (CNRS), Riad al Soloh, 1107 2260, Beirut, Lebanon*
(Dated: July 11, 2017)

We begin by developing a global structural understanding of the Lebanese power grid that reveals a certain level of decentralization via numerous strongly connected components. Then, using Apache Spark, we address its topological vulnerability subject to random and cascading failures by which energy centers in Lebanon can be exposed and are at risk. Subsequently, a spatial analysis of the failures is presentend. Our Spark implementation achieves significant speedup on ? cores and concludes in real time for a graph with about $800,000$ nodes. The amenability of our work to big data processing makes it extendable to larger networks of networks, towards a fuller understanding of resilience at many vital levels beyond the power grid.

PACS numbers: 89.75.-k, 89.65.Lm, 89.75.Kd, 88.40.fc

I. INTRODUCTION

In scale-free networks (SF) the probability of a node being connected to k others exhibits a power-law distribution $P(k) \propto k^{-\alpha}$, which is a topological property affecting and controlling their resilience or the measure of their functionality subject to disruptions [1–5]. Examples of this class of SF are the Internet, power systems, and transportations networks, which are real-world networks shown to be robust when subject to random failures however, display a high vulnerability when prone to cascading ones [6–9].

In power systems, and unlike random failures which emerge locally, blackouts are severe events associated with cascading behavior leading to global network collapse [10–16]. Examples such as the one affecting the north-east in the US and eastern Canada in 2003 burdened their economies with 10 billions dollars of direct costs [13]. Such failures can be linked to either structural dependencies, where the damage spreads via structurally dependent connections, or functional overloads, where the flow goes through alternative paths leading to overloaded nodes. Thus understanding the propagation of these failures becomes pivotal in developing and deploying protective and mitigating strategies.

Large power systems and real-world networks in general exhibit an exponentially increasing combinatorial number of failure nodes. This levies an overwhelming computational burden that cannot be accomplished in real-time. In several leading works such as [1, 17, 18], the power grid, particularly the network of its transmission lines, is analyzed using graph algorithms such as betweenness centrality and shortest paths. In this paper, we build on the approach in [1] using an Apache Spark implementation of topological vulnerability analysis of the Lebanese power grid subject to random and cascading failures. Beyond the scholarly aspects of our proposed work, our analysis contributes towards precision-based policy making and disaster response in a region marred by wars and under-development. Using elements from distributed algorithm design, abstract data structures, as well as spatial analysis, our work extends the approach of [1] in analyzing the vulnerability of transmission lines to the assessment of that of the whole Lebanese power grid including its generation, distribution, and transmission lines.

Our manuscript is organized as follows. In Sec. II, we present an overview of related work that tackles power grid resilience analysis as well as implementations of it that runs on distributed systems, and we describe some intrinsics related to Apache spark for big data distributed processing. In Sec. III, we present our distributed graph algorithm that builds on the loosely centralized structure of the Lebanese power grid using Breadth First Search and Vertex Betweenness Centrality, orchestrated using four scenarios known in the literature [1]. Each of these scenarios simulates a unique temporal mode of removal of vertices. In Sec. IV, we perform run-time analysis of our algorithm and obtain excellent scalability as the number of processing cores increases up to 32, a result which we attribute to the large number of SCCs found in the Lebanese power grid and the fact that majority of SCCs have a relatively small cardinality. We demonstrate the loss in connectivity in the power grid associated with each scenario. We also perform a spatial correlation analysis of failures using the geocoding of vertices that have led substantial loss in connectivity. In Sec. V, we conclude with remarks around the impact of our work.

II. BACKGROUND

A. Resilience analysis using Graph Theory

Network connectivity is closely tied to the notion of betweenness centrality. Given an arbitrary vertex in a graph, its betweenness centrality is defined to be the number of shortest paths that pass through the given vertex. Exploiting betweenness centrality has been widespread in a number of works addressing power grid vulnerability analysis [1, 17]. The work in [1] employs this notion using four scenarios each of which simulates a unique temporal mode of removal of vertices. For example, the overall connectivity of the graph is re-examined upon removal of transmission nodes according to the following orders: (1) totally random order (2) decreasing order of node degrees (load) (3) decreasing order of node betweenness centrality and (4) decreasing order of node betweenness centrality in cascading order resulting from the nodes' dynamical removal. Our Spark implementation addresses the random and cascading failure scenarios explored by [1].

B. Distributed Computation frameworks

C. MapReduce and Pregel

The last few years have witnessed an uptake in distributed data processing research. Among the leading frameworks to exploit distributed computation on commodity hardware is the MapReduce paradigm [25]. A typical MapReduce program consists of the "Map" operator that parcels out work to various nodes within the cluster or map, and the "Reduce" phase that applies a reduction operator on the results from each node into a global query. The key contributions of the MapReduce framework are the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine, for example, by reassigning tasks when a given execution fails. As with all other parallel and distributed paradigms, the performance of an efficient MapReduce algorithm is contingent upon a reduced communication cost. Of particular challenge is how to efficiently process large graphs. Graph algorithms often exhibit poor locality of reference, and a low compute-to-memory access ratio, which affects the scalability of their parallel adaptations. It is also difficult to maintain a steady degree of parallelism over the course of execution of graph algorithms. Additionally, expressing a graph algorithm in MapReduce requires passing the entire state of the graph from one stage to the next, thus imposing significant communication as well as serialisation in the parallel code.

The first serious development for supporting graph algorithms using the MapReduce framework is found in Google's Pregel [26]. Instead of coordinating the steps of a chained MapReduce program, Pregel is able to process iteration over supersteps under the Bulk Synchronous Parallel model [27–29]. In a BSP algorithm, a computation proceeds in a series of global supersteps. Each superstep consists of three phases:

1. A concurrent computation superstep: here, each processor performs local computations using values stored in the local, fast memory of the processor.
2. A communication superstep: here, the processes exchange data between themselves if needed for the aggregation of the results computed in (1) above.
3. A barrier synchronisation superstep: here, each processor arriving at this point waits until all other processes have reached the same barrier.

According to this model, a graph algorithm in Pregel is organised as a sequence of iterations, and can be described from the point of view of a vertex, that manages its state and sends messages only to its neighbours. Pregel keeps vertices and edges on the machine that performs computation, and uses network transfers only for messages.

D. Spark and GraphX

Spark, a distributed computation framework built around the MapReduce paradigm, is a recent Apache foundation software project supported by an execution engine for big data processing. Spark provides for in-memory computation, which refers to the storage of information in the main random access memory (RAM) of dedicated servers rather than in relational databases running on relatively slower disk drives. Using over 80 high-level operators, Spark makes it possible to write code more succinctly, and till this time, is considered one of the fastest frameworks for big data processing. Spark's most notable properties are also thanks to its core, which, in addition for serving as the base engine for large-scale parallel and distributed data processing, is able to handle memory management and fault recovery, scheduling, distributing and monitoring jobs on a cluster, as well as interacting with storage systems.

Spark hinges on parallel abstract data collections called RDDs (resilient distributed datasets), which can distributed across a cluster. These RDDs are immutable, partitioned data structures that can be manipulated through multiple operators like Map, Reduce, and Join. For example, RDDs are created through parallel transformations (e.g., map, group by, filter, join, create from file systems). RDDs can be cached (in-memory) by allowing to keep data sets of interest locally across operations, thus contributing to a substantial speedup. At the same time, Spark uses lineage to support fault tolerance, i.e., record all the operations/transformations that yielded RDDs from a source data. In case of failure, an RDD can be reconstructed given the transformation functions contributing to that RDD. Additionally, after creating RDDs, it is possible to analyse them using actions such as count, reduce, collect and save. Note that all operations/transformations are lazy until one runs an action. At that point, the Spark execution engine pipelines operations and determines an execution plan.

Borrowing from Pregel, GraphX [30] is a platform built on top of Spark that provides APIs for parallel and distributed processing on large graphs. In GraphX, each graph is mapped into different RDDs, where in each RDD one applies the computation on the graph using the think like a vertex model.

III. MATERIALS AND METHODS

We simulate the failures by node removal sorted in a random or a particular order given by the nodes with the highest betweenness, nodes with the highest betweenness in cascading order resulting from their dynamical removal, the nodes with the highest loads.

Given an undirected graph $G = (V, E)$, we define its connectivity as follows:

$$\texttt{connectivity}(G) = \sum_{v \in V} \mid \texttt{reachable}(v) \mid,$$

where $\texttt{reachable}(v)$ is all the reachable nodes from v.

Given a node x, we define the loss of a graph G with respect to x as follows:

$$\texttt{loss}(G, x) = \texttt{connectivity}(G) - \texttt{connectivity}(G \setminus x)$$

where $G \setminus x$ is a graph defined by removing vertex x in G and all its edges.

We denote by $\texttt{SCC}(G) = \{G_1, \ldots, G_n\}$ to be the maximal strongly connected components of G. We have $\texttt{loss}(G, x) = \texttt{loss}(G_i, x) + \sum_{j \neq i} \texttt{connectivity}(G_j)$, where $x \in V_i$, and $\texttt{connectivity}(G_j) = \mid V_j \mid \times \mid V_j - 1 \mid$.

```
attackGraph(Graph graph) {
   for(i = 0 until |V|) {
      select victim vertex v
      remove vertex v
      update loss with respect to v
   }
}
```

We select a victim vertex according to one of the following four scenarios:

- *Random*: a random vertex is selected.
- *Degree-based*: the vertex with the highest degree is selected.
- *Betweenness Centrality*: the vertex with the highest betweeness centrality is selected. The betweenss centrality of the nodes is only computed once (i.e., it will not be updated after removing a vertex).
- *Cascanding*: Similar to the betweeness centrality scenario, however, the betweeness centrality of the nodes is updated at each iteration (i.e., after removal of a victim vertex).

Given a graph G, the betweeness centrality of a node v is equal to $\texttt{bc}_G(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v. Clearly, we have $\texttt{bc}_G(v) = \texttt{bc}_{G_i}(v)$, where $\texttt{SCC}(G) = \{G_1, \ldots, G_n\}$ and $v \in G_i$.

A. Spark-based Implementation

We provide an efficient Spark-based implementation of the all the above scenarios. Spark allows to parallelize and distributed computations on several nodes. Spark [**?**] is new programming model supported by an execution engine for big data processing. Spark is based on RDD (Resilient Distributed Data Set). RDDs are big parallel collections that can be distributed across a cluster. RDDs are created through parallel transformations (e.g., map, group by, filter, join, create from file systems). Moreover, RDDs can be cached (in-memory) by allowing to keep data sets of interest in Memory across operations and thus contribute to a substantial speedup. At the same time, Spark uses lineage to support fault tolerance, i.e., record all the operations/transformations that yield to create RDDs from a source data. That is, in case of failure, an RDD can be reconstructed given the transformation functions yielding to that RDD. Additionally, after creating RDDs, it is possible to do analytics on them by running actions on them such as count, reduce, collect and save. Note that all operations/transformations are lazy until you run an action. Then, the Spark execution engine pipelines operations and finds an execution plan. GraphX [30] is a platform built on top of Spark that provides APIs for parallel and distributed processing on large graphs. The spark-based implementation is done as follows:

- Given a file (read from local or distributed file system, i.e., HDFS) containing information about the graph and a number of partitions, we build the corresponding graph RDD.
- We compute the strongly connected components on the graph RDD.
- We create an RDD, `rddSCC` where each item corresponds to a strongly connected component.
- On each item `rddSCC` (i.e., each strongly connected component), we iteratively identifies victim nodes (and locally update the corresponding component) until all the nodes are identified. The output of each step produces an array per item containing the losses in a decreasing order that correspond to the nodes of that item. Note that, the loss is computed and affected locally on each item.
- We define a reduce operation that merge-sorts all the generated arrays and sequentially removes the nodes in a decreasing order. The output of this step produces one array containing the global losses (in a decreasing order) corresponding to all the nodes.

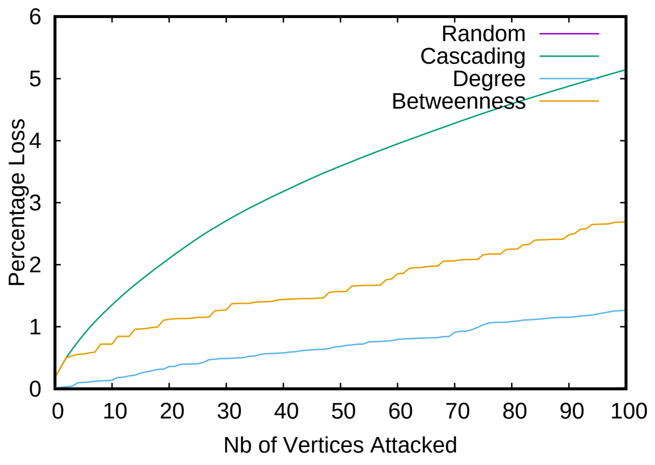

FIG. 1. Loss first 100. todo

Global losses are computed by by accumulating the sum of all the previous items.

IV. RESULTS

Our own Spark implementation of contingency analysis executes in real-time for the entire power grid, achieving a speed-up of ? on ? processors. We conclude with a spatial understanding of the hotspots on Lebanese soil where energy centers can be exposed and are at risk, using a spatial correlation supported by a binary search tree. The amenability of our work to big data processing makes it extendable to larger networks of networks, towards a fuller understanding of resilence at many vital levels beyond the power grid. Examples are as communications network, Internet networks, transportation networks, hospitals and medical centers networks, to name a few.

TODO - discuss the benchmarks:

- Property of the graph (number of components, node per component)
- For each scenario we vary the number of threads and partitions.
- For each scenario we compute the loss.
- Scalability - Replicate the graph:

V. CONCLUSION

Contingency analysis is a security function to assess the ability of a power grid to sustain various combinations of power grid component failures at energy control centers. To date, there exists no such work to examine the power grid resilience in Lebanon, a country which is

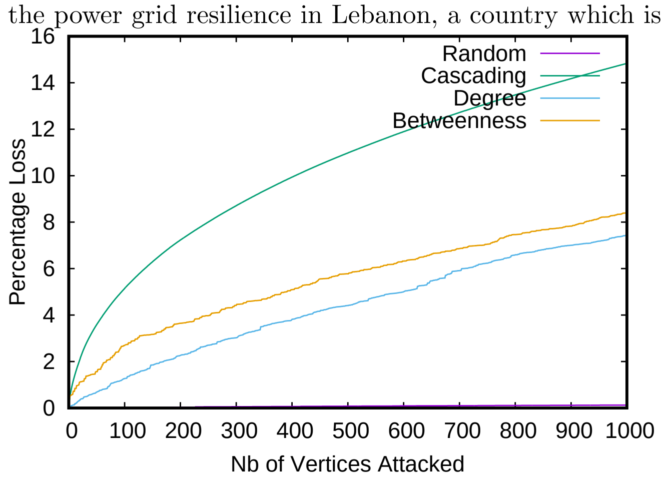

FIG. 2. Loss first 1000.. todo

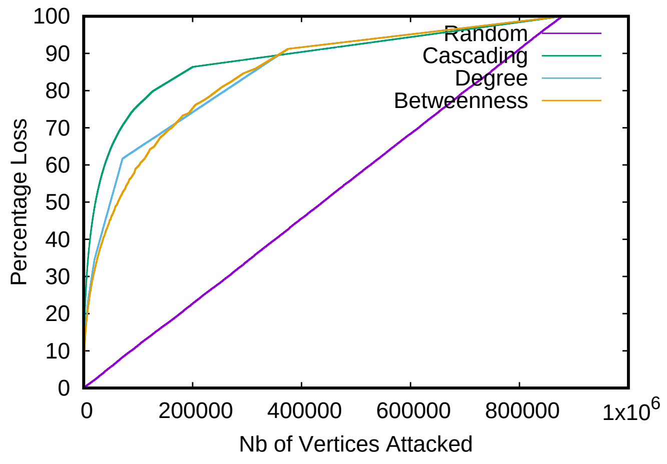

FIG. 3. Loss all... to do

still reeling under the effect of a brutal civil war, and recently, bearing an additional burden associated with the spillover from the Syrian war. This neighbouring conflict has exposed Lebanon to a number of random terrorist attacks, and caused it to become one of the major hosts of Syrian refugees, in addition to Iraqi and Palestinian refugees in former years. The strains on Lebanon's vital infrastructure have also been affecting the host community itself, where electric power supply is becoming increasingly drained. Coupled with a severe shortage of funds that can help restructure the entire system, the Lebanese power grid has become plagued with nationwide blackouts, making it an unreliable source of electricity generation, transmission, and distribution.

TABLE I. graph1.. todo

Threads	BC-4	BC-8	BC-16	BC-32	C-4	C-8	C-16	C-32	D-4	D-8	D-16	D-32	R-4	R-8	R-16	R-32
64	27	17	22	41	53	34	32	52	26	15	20	41	23	17	20	39
32	24	17	19	37	52	33	31	49	20	16	18	36	20	19	19	36
16	24	18	22	39	53	35	37	54	21	16	19	37	19	16	20	37
8	25	25	30	54	55	50	56	80	21	21	26	51	20	20	26	50
4	41	41	53	97	87	86	98	138	33	33	45	89	30	31	44	90
2	82	84	108	206	165	165	189	287	65	66	92	188	55	62	87	186
1	161	167	219	423	312	320	369	543	124	127	187	384	104	114	174	344

TABLE II. graph 2.. todo

Threads	BC-4	BC-8	BC-16	BC-32	C-4	C-8	C-16	C-32	D-4	D-8	D-16	D-32	R-4	R-8	R-16	R-32
64	127	71	48	55	119	64	60	76	108	58	45	53	45	32	31	50
32	120	59	45	49	116	68	56	71	108	57	42	47	40	29	29	45
16	128	66	39	52	115	66	65	79	112	57	36	50	41	33	30	46
8	123	66	56	77	120	97	98	121	106	54	46	68	40	34	39	63
4	164	104	101	142	210	191	185	223	134	83	80	119	60	57	69	113
2	263	203	207	292	416	353	366	462	209	159	153	239	111	110	131	224
1	452	357	371	525	761	804	793	969	355	281	286	434	196	201	230	402

[1] R. Albert, H. Jeong, and A.-L. Barabási, Nature **406**, 378 (2000).

[2] M. E. J. Newman, SIAM Review **45**, 167 (2003).

[3] J. Gao, X. Liu, D. Li, and S. Havlin, Energies **8**, 12187 (2015).

[4] A. Bashan, Y. Berezin, S. V. Buldyrev, and S. Havlin, Nature Publishing Group **9**, 1 (2013).

[5] J. Gao, X. Liu, D. Li, and S. Havlin, Energies **8**, 12187 (2015).

[6] E. Bompard, D. Wu, and F. Xue, Electric Power Systems Research **81**, 1334 (2011).

[7] L. Dueñas-Osorio and S. M. Vemuru, Structural Safety **31**, 157 (2009).

[8] D. Manik, M. Rohden, H. Ronellenfitsch, X. Zhang, S. Hallerberg, D. Witthaut, and M. Timme, arXiv.org , arXiv:1609.04310 (2016), 1609.04310.

[9] R. Cohen, K. Erez, D. ben Avraham, and S. Havlin, Physical Review Letters **86**, 3682 (2001).

[10] M. Rosas-Casals, S. Valverde, and R. V. Solé, I. J. Bifurcation and Chaos (2007).

[11] E. Bompard, R. Napoli, and F. Xue, International Journal of Critical Infrastructure Protection **2**, 5 (2009).

[12] C. D. Brummitt, P. D. H. Hines, I. Dobson, C. Moore, and R. M. D'Souza, Proceedings of the National Academy of Sciences of the United States of America **110**, 12159 (2013).

[13] L. Daqing, J. Yinan, K. Rui, and S. Havlin, Nature Scientific Reports **4**, 1 (2014).

[14] R. Albert, I. Albert, and G. L. Nakarado, Physical Review E **69**, 025103 (2004).

[15] J.-W. Wang and L.-L. Rong, Safety Science **49**, 807 (2011).

[16] R. V. Solé, M. Rosas-Casals, B. Corominas-Murtra, and S. Valverde, Physical Review E **77**, 026102 (2008).

[17] S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. Chung Wong, in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2010) pp. 1–7.

[18] L. Daqing, J. Yinan, K. Rui, and S. Havlin, Scientific Reports **4**, 5381 (2014).

[19] R. Albert, H. Jeong, and A.-L. Barabási, Nature (London) **406**, 378 (2000).

[20] R. Cohen, D. ben Avraham, and S. Havlin, Phys. Rev. Lett. **85**, 4626 (2000).

[21] R. Cohen, D. ben Avraham, and S. Havlin, Phys. Rev. Lett. **86**, 5468 (2001).

[22] D. Callaway, M. Newman, S. Strogatz, and D. Watts, Phys. Rev. Lett. **85**, 5468 (2000).

[23] A. Cavagna, A. Cimarelli, I. Giardina, G. Parisi, R. Santagati, F. Stefanini, and M. Viale, Proc. Natl. Acad. Sci. **107**.

[24] H. A. Makse, S. Havlin, and H. E. Stanley, Nature **377**.

[25] J. Dean and S. Ghemawat, Comm. ACM **51**.

[26] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, in *SIGMOD 2010* (2010) pp. 135–146.

[27] R. H. Bisseling, *Parallel Scientific Computation* (Oxford University Press, 2004).

[28] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudrea, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, Parallel Computing **24**.

[29] L. G. Valiant, Communications of the ACM **33**.

[30] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* (2014) pp. 599–613.

TABLE III. graph 4.. todo

Threads	BC-4	BC-8	BC-16	BC-32	C-4	C-8	C-16	C-32	D-4	D-8	D-16	D-32	R-4	R-8	R-16	R-32
64	525	317	171	117	411	179	127	123	430	220	134	104	133	79	70	87
32	491	274	153	99	387	202	123	121	449	243	149	100	122	101	64	75
16	473	311	133	96	401	184	127	137	457	236	113	77	130	86	62	78
8	557	250	132	126	387	234	204	212	452	222	113	105	119	93	75	93
4	750	333	221	234	612	431	374	398	601	274	168	190	151	126	120	163
2	1041	526	430	467	1042	791	736	793	888	436	344	395	273	235	237	337
1	1626	902	879	1021	2156	1784	1577	1754	1584	905	730	783	533	458	456	647