NScaleSpark: Subgraph-centric Graph Analytics on Apache Spark

Abdul Quamar
University of Maryland
abdul@cs.umd.edu

Amol Deshpande
University of Maryland
amol@cs.umd.edu

ABSTRACT

In this paper, we describe NSCALESPARK, a framework for executing large-scale distributed graph analysis tasks on the Apache Spark platform. NSCALESPARK is motivated by the increasing interest in executing rich and complex analysis tasks over large graph datasets. There is much recent work on *vertex-centric graph programming frameworks* for executing such analysis tasks – these systems espouse a "think-like-a-vertex" (TLV) paradigm, with some example systems being Pregel, Apache Giraph, GPS, Grace, and GraphX (built on top of Apache Spark). However, the TLV paradigm is not suitable for many complex graph analysis tasks that typically require processing of information aggregated over neighborhoods or subgraphs in the underlying graph. Instead, NSCALESPARK is based on a "think-like-a-subgraph" paradigm (also recently called "think-like-an-embedding" [23]). Here, the users specify computations to be executed against a large number of multi-hop neighborhoods or subgraphs of the data graph. NSCALESPARK builds upon our prior work on the NSCALE system [18], which was built on top of the Hadoop MapReduce system. We describe how we reimplemented NSCALE on the Apache Spark platform, the key challenges therein, and the design decisions we made. NSCALESPARK uses a series of RDD transformations to extract and hold the relevant subgraphs in distributed memory with minimal footprint using a cost-based optimizer. Our in-memory graph data structure enables efficient graph computations over large-scale graphs. Our experimental results over several real world data sets and applications show orders-of-magnitude improvement in performance and total cost over GraphX and other vertex-centric approaches.

1. INTRODUCTION

Increasing interest in executing complex analysis tasks over large graphs has led to much recent work on large-scale distributed graph processing frameworks. However, because of a wide variety and range of graph analysis tasks and algorithms that are of interest, there is little consensus on the programming paradigms or abstractions around which to build such large-scale graph processing systems. Perhaps the most popular such paradigm is the *vertex-centric programming framework* (also called "think-like-a-vertex" (TLV) paradigm). In these frameworks, users write vertex-level *compute* programs, that are then executed iteratively by the framework in either a bulk synchronous fashion or asynchronous fashion using message passing or shared memory. This model is well-suited for some graph processing tasks like computing PageRank or connected components, and also for several distributed machine learning and optimization tasks that can be mapped to message passing algorithms in appropriately constructed graphs [13, **?**]. Originally introduced in this context in Google's Pregel system [14], several graph analytics systems are built around this model (e.g., Apache Giraph, Hama, GraphLab [13], PowerGraph, GRACE [25], GPS [19], GraphX [8]).

Limitations of TLV paradigm: However, this model limits the compute program's access to a single vertex's state and so the overall computation needs to be decomposed into smaller local tasks that can be (largely) independently executed; it is not clear how to do this for many graph algorithms of interest, without requiring a large number of iterations. For example, to analyze a 2-hop neighborhood around a vertex to find social circles [15], one would first need to gather all the information from the 2-hop neighbors through message-passing, and reconstruct those neighborhoods locally (i.e., in the local states of the vertex programs). Even something as simple as computing the number of triangles for a node requires gathering information from 1-hop neighbors (since we need to reason about the edges between the neighbors). This requires significant network communication and an enormous amount of memory.

Consider some back-of-the-envelope calculations for estimating the message passing and memory overheads for constructing neighborhoods of various sizes at each vertex. Here we use a sample of the Orkut social network graph, containing approximately 3M nodes, 234M edges and an average degree of 77. The original graph occupies 14GB of memory for a data structure that stores the graph as a bag of vertices in adjacency list format. Constructing the 1-hop neighborhoods for all vertices through message passing requires 231M messages and consumes a total of 233 GB of cluster memory, whereas constructing 2-hop neighborhoods would require approximately 18B messages and 18TB of memory. It is clear that a vertex-centric approach requires inordinate amounts of network traffic, beyond what can be addressed by "combiners" in Pregel or GPS, and impractical amount of cluster memory. Although GraphLab is based on a shared memory model, it too would require two phases of GAS (Gather, Apply, Scatter) to construct a 2-hop neighborhood at each vertex and also suffers from duplication of state and high memory overhead. Furthermore, because most existing graph processing frameworks hash-partition vertices

NDA'16 , June 26-July 01 2016, San Francisco, CA, USA

 ISBN 978-1-4503-4513-2/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2980523.2980529

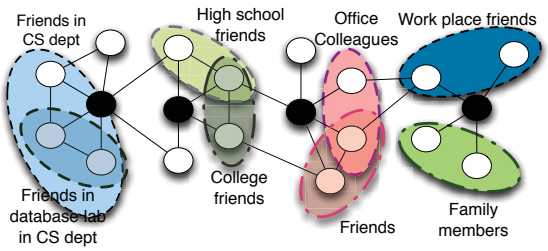

Figure 1: An example of neighborhood-centric analysis: identify users' *social circles* in a social network.

by default, this approach will create much duplication of neighborhood data structures. In recent work, Seo et al. [21] also observe that these frameworks quickly run out of memory and do not scale for ego-centric analysis tasks.

NSCALESPARK: In this paper, we present NSCALESPARK, which supports a significantly more general subgraph-centric programming framework that we introduced in our prior work [18], while keeping many of the benefits of the vertex-centric programming frameworks including the ability to parallelize the task across a distributed cluster of machines. NSCALESPARK is better suited for a large class of graph analysis tasks that can be viewed as operations on local neighborhoods (or subgraphs of local neighborhoods) of a large number of nodes in the graph. For example, there is much interest in analyzing *ego networks*, i.e., 1- or 2-hop neighborhoods, of the nodes in the graph. Examples of specific ego network analysis tasks include identifying *structural holes*, brokerage analysis, counting *motifs* [16], identifying *social circles* (Figure 1) [15], link prediction and recommendations using Personalized Page Rank [6], computing *local clustering coefficients*, and anomaly detection [5]. In other cases, there may be interest in analyzing connected or induced subgraphs satisfying certain properties. As an example, we may be interested in analyzing the induced subgraph on users who tweet a particular *hashtag* in the Twitter network. Similarly, we may be interested in analyzing groups of users who have exhibited significant communication activity in recent past. More complex subgraphs can be specified as *unions* or *intersections* of neighborhoods of pairs of nodes; this may be required for graph cleaning tasks such as *link prediction* and *entity resolution*.

More specifically, in NSCALESPARK, the user specifies[1]: (a) the subgraphs of interest as k-hop neighborhoods around vertices or sets of vertices that satisfy a set of predicates, and (b) a user program to be executed on those subgraphs (which may itself be iterative). The user program is written against a general graph API (specifically, BluePrints), and has access to the entire state of the subgraph against which it is being executed. The NSCALESPARK execution engine is in charge of ensuring that the user program only has access to that state and nothing more; this guarantee allows existing graph algorithms to be used without modification. Thus a program written to compute, say, connected components in a graph, can be used as is to compute the connected components within each subgraph of interest.

NSCALESPARK consists of two major components. First, the graph extraction and packing (GEP) module extracts relevant subgraphs of interest and uses a cost-based optimizer for data replication and placement that minimizes the number of machines needed, while attempting to balance load across machines to guard against the straggler effect. Second, the distributed execution engine executes user-specified computation on the subgraphs in memory. It employs several optimizations that reduce the total memory footprint by exploiting overlap between subgraphs loaded on a machine, without compromising correctness.

Whereas NSCALE was built on top of Hadoop, NSCALESPARK is written using Apache Spark [26], which has emerged as a popular big data analytics platform in the recent years. It provides the unique ability to prune large datasets through a series of coarse-grained transformations and to hold them in distributed memory for further analysis providing great performance benefits, especially for iterative analysis tasks. It uses a lineage graph to achieve fault tolerance without resorting to intermediate state materialization as is done in Hadoop MapReduce. As such, it is a viable platform for big data data analytics which provides transparent distribution of data and computation and fault tolerance at scale.

Although, Spark provides its own graph analytics library, GraphX, it does not scale well for many neighborhood- or subgraph-centric analysis tasks as we showed in our prior work (and often does not finish even for small graphs). It basically emulates the vertex-centric programming frameworks thus suffering from the same limitations. The storing of the vertex and edge information as separate immutable RDDs further aggravates the problem of aggregating neighborhood state and executing user computation on it. Our experimental results show that NSCALESPARK provides significant improvement over GraphX both in terms of performance and resource consumption (memory required).

Related Work: There have been several other proposals for subgraph-centric programming frameworks. Giraph++ [24] and GoFFish [22] both primarily target the message passing overheads and scalability issues in the vertex-centric, BSP model of computation. Giraph++ partitions the graph onto multiple machines, and runs a sequential algorithm on the entire subgraph in a partition in each superstep. GoFFish is very similar and partitions the graph using METIS (another scalability issue) and runs a connected components algorithm in each partition. An important distinction is that in both cases, the subgraphs are determined by the system, in contrast to our framework, which explicitly allows users to specify the subgraphs of interest. Furthermore, these previous frameworks use serial execution within a partition and the onus of parallelization is left to the user. It would be extremely difficult for the end user to incorporate tools and libraries to parallelize these sequential algorithms to exploit powerful multicore architectures available today.

Arabesque [23], proposed recently, is much closer to our work, and also espouses a "think-like-an-embedding" paradigm. Their focus is primarily on graph mining tasks like frequent pattern mining where the intermediate result sizes could be larger than the input graph itself, and they assume that the entire graph can be maintained in memory at each of the cluster nodes. In this work, we assume that the graph is too large for any single machine and must be partitioned across the machines; in ongoing work, we are investigating the issues in bridging the gap between these two models.

Among other graph programming frameworks, SociaLite [20] describes an extension of a Datalog-based query language to express graph computations such as PageRank, connected components, shortest path, etc. The system uses an underlying relational database with tail-nested tables and enables users to hint at the execution order. Galois [17], LFGraph [10], are among highly scalable general-purpose graph processing frameworks that target systems-

[1]This has subtle differences from the NSCALE programming framework as we discuss later.

or hardware-level optimization issues, but support only low-level or vertex-centric programming frameworks. Facebook's Unicorn system [7] constructs a distributed inverted index and supports online graph-based searches using a programming API that allows users to compose queries using set operations like AND, OR, etc.; thus Unicorn is similar to an online SPARQL query processing system and can be used to identify nodes or entities that satisfy certain conditions, but it is not a general-purpose complex graph analytics system. We omit a more detailed discussion of other systems (e.g., XStream, GraphChi) due to lack of space.

Outline: We begin with a brief overview of our prior NSCALE system (Section 2). We then describe NSCALESPARK, the key challenges we faced in porting NSCALE to Spark, and some of the design decisions that we made (Section 3). We then present an experimental evaluation comparing NSCALESPARK with NSCALE and GraphX (Section 4).

2. BACKGROUND: NSCALE

In this section, we briefly summarize the NSCALE system that we proposed in our prior work, including some of the representative applications that motivated that system. A comprehensive experimental evaluation comparing NSCALE with Apache Giraph, GraphLab, and GraphX showed that NSCALE performs 3X to 30X better that those systems for analysis tasks over 1-hop neighborhoods and 20X to 400X for 2-hop neighborhood analytics.

2.1 Application Scenarios

We begin with discussing two representative graph analysis tasks that are ill-suited for vertex-centric frameworks, that we use for our experimental evaluation in this paper. Several other applications are described in detail in [18].

Local clustering coefficient (LCC). In a social network, the LCC quantifies, for a user, the fraction of his or her friends who are also friends—this is an important starting point for many graph analytics tasks. Computing the LCC for a vertex requires constructing its ego network, which includes the vertex, its 1-hop neighbors, and all the edges between the neighbors. Even for this simple task, the limitations of vertex-centric approaches are apparent, since they require multiple iterations to collect the ego-network before performing the LCC computation (such approaches quickly run out of memory as we increase the number of vertices we are interested in).

Social recommendations using personalized PageRank. Random walks with restarts (such as personalized PageRank (PPR) [6]) lie at the core of several social recommendation algorithms. These algorithms can be implemented using Monte-Carlo methods [9] where the random walk starts at a vertex v, and repeatedly chooses a random outgoing edge and updates a visit counter with the restriction that the walk jumps back only to v with a certain probability. The stationary distribution of such a walk assigns a PageRank score to each vertex in the neighborhood of v; these provide the basis for link prediction and recommendation algorithms. Implementing random walks in a vertex-centric framework would involve one iteration with message passing for each step of the random walk. In contrast, with NSCALESPARK the complete state of the k-hop neighborhood around a vertex is available to the user's program, which can then directly execute personalized PageRank or any existing algorithm of choice.

2.2 Programming Model

We assume a standard definition of a graph $G(V, E)$ where $V = \{v_1, v_2, ..., v_n\}$ denotes the set of vertices and $E = \{e_1, e_2, ..., e_m\}$ denotes the set of edges in G. Let $A = \{a_1, a_2, ..., a_k\}$ denote the union of the sets of attributes associated with the vertices and edges in G. In contrast to vertex-centric programming models, NSCALE allows users to specify subgraphs or neighborhoods as the scope of computation. More specifically, users need to specify: (a) subgraphs of interest on which to run the computations through a *subgraph extraction query*, and (b) a user program.

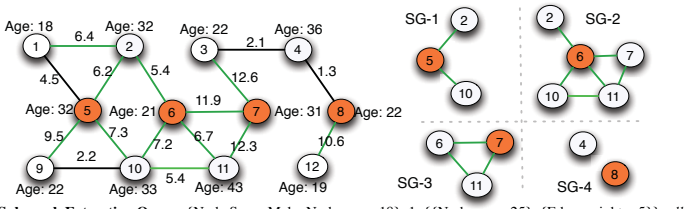

Figure 2: A subgraph extraction query on a social network

```
ArrayList<RVertex> n_arr = new ArrayList<RVertex>();
for(Edge e: this.getQueryVertex().getOutEdges)
     n_arr.add(e.getVertex(Direction.IN));

int possibleLinks = n_arr.size()* (n_arr.size()-1)/2;

// compute #actual edges among the neighbors
for(int i=0; i < n_arr.size()-1; i++)
   for(int j=i+1; j < n_arr.size(); j++)
      if(edgeExists(n_arr.get(i), n_arr.get(j)))
          numEdges++;
double lcc = (double) numEdges/possibleLinks;
```

Figure 3: Example user program to compute *local clustering coefficient* written using the BluePrints API. The *edgeExists()* call requires access to neighbors' states, and thus this program cannot be executed as is in a vertex-centric framework.

Specifying subgraphs of interest. NSCALE supported extraction queries that are specified in terms of four parameters: (1) a predicate on vertex attributes that identifies a set of *query vertices* (P_{QV}), (2) k – the radius of the subgraphs of interest, (3) edge and vertex predicates to select a subset of vertices and edges from those k-hop neighborhoods (P_E, P_V), and (4) a list of edge and vertex attributes that are of interest (A_E, A_V). This captures a large number of subgraph-centric graph analysis tasks, including all of the tasks discussed earlier. For a given subgraph extraction query q, we denote the subgraphs of interest by $SG_1(V_1, E_1), ..., SG_q(V_q, E_q)$.

Figure 2 shows an example subgraph extraction query, where the query vertices are selected to be vertices with $age > 18$, radius is set to 1, and the user is interested in extracting induced subgraphs containing vertices with $age > 25$ and edges with $weight > 5$. The four extracted subgraphs, $SG_1, ..., SG_4$ are also shown.

Specifying subgraph computation user program. The user computation to be run against the subgraphs is specified as a Java program against the BluePrints API [1], a collection of interfaces analogous to JDBC but for graph data. Blueprints is a generic graph Java API used by many graph processing and programming frameworks (e.g., Gremlin, a graph traversal language [3]; Furnace, a graph algorithms package [2]; etc.). By supporting the Blueprints API, we immediately enable use of many of these already existing toolkits over large graphs. Figure 3 shows a sample code snippet of

how a user can write a simple local clustering coefficient computation using the BluePrints API. The subgraphs of interest here are the 1-hop neighborhoods of all vertices (by definition, a 1-hop neighborhood includes the edges between the neighbors of the node).

2.3 System Overview

NSCALE has two main components.

Graph Extraction and Packing (GEP) Module. Unlike prior graph processing frameworks, the GEP module forms a major component of the overall NSCALE framework. From a usability perspective, it is important to provide the ability to read the underlying graph from the persistent storage engines that are not naturally graph-oriented. However, more importantly, partitioning and replication of the graph data are more critical for graph analytics than for analytics on, say, relational or text data.

Graph analytics tasks, by their very nature, tend to traverse graphs in an arbitrary and unpredictable manner. If the graph is partitioned across a set of machines, then many of these traversals are made over the network, incurring significant performance penalties. Further, as the number of partitions of a graph grows, the number of *cut* edges (with endpoints in different partitions), and hence the number of distributed traversals, grows in a non-linear fashion. This is in contrast to relational or text analytics where the number of machines used has a minor impact on the execution cost.

NSCALE treats user programs as black-boxes and avoids distributed traversals altogether by replicating vertices and edges sufficiently so that every subgraph of interest is fully present in at least one partition. Similar approach has been taken by some of the prior work such as SPARQL queries [11] in distributed settings. The GEP module is used to ensure this property, and is responsible for extracting the subgraphs of interest and packing them onto a small set of partitions such that every subgraph of interest is fully contained within at least one partition.

NSCALE uses Apache Hadoop Map-Reduce platform for implementing the GEP phase, and we refer the reader to [18] for further details on that implementation.

Distributed Execution Engine. The distributed execution phase in NSCALE is implemented as a MapReduce job, which reads the original graph and the mappings generated by GEP, shuffles graph data onto a set of reducers, each of which constructs one of the partitions. Inside each reducer, an instance of the execution engine is instantiated along with the user program, which then receives and processes the graph partition.

The NSCALE execution engine supports both serial and parallel execution modes for executing user programs on the extracted subgraphs. For serial execution, the execution engine uses a single thread and loops across all the subgraphs in a partition, whereas for parallel execution, it uses a pool of threads to execute the user computation in parallel on multiple subgraphs in the partition. However, this is not straightforward because the different subgraphs of interest in a partition are stored in an overlapping fashion in memory to reduce the total memory requirements. The execution engine employs several bitmap-based techniques [18] to ensure correctness in that scenario.

3. NSCALESPARK SYSTEM DESIGN

In this section, we discuss in detail the various aspects of the NSCALESPARK system design. We first list out the major challenges in the NSCALESPARK system design and then provide a brief overview of the system architecture. Figure 4 shows the high-level system architecture of NSCALESPARK. The framework supports ingestion of the underlying graph in a variety of different formats including edge lists, adjacency lists, and in a variety of different types of persistent storage engines including key-value pairs, specialized indexes stored in flat files, relational databases, etc. As in NSCALE, the two major components of NSCALESPARK are the graph extraction and packing (GEP) module and the distributed execution engine embedded as a library within the the Spark runtime environment.

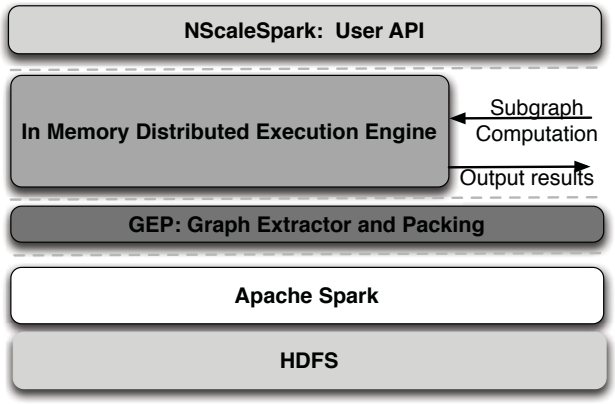

Figure 4: High-level System Architecture

Some of the key challenges that needed to be addressed in designing and building NSCALESPARK include:

- Providing an efficient mechanism for extracting the user-defined subgraphs from the underlying raw graph data using a series of coarse-grained transformations supported by the Spark API.
- Designing and developing an appropriate abstraction for holding the extracted subgraphs in a distributed setting while minimizing the memory footprint for the same.
- Building an efficient execution model that would execute the user computation on the extracted subgraphs in distributed memory without incurring the overheads of the vertex-centric approaches.
- Providing support for both one pass and iterative analytics while keeping in mind the limitations of the Spark execution model arising due to the immutability of the RDDs.

NSCALESPARK is still under active development, and in this section, we describe the functionality that NSCALESPARK is envisioned to support.

Programming Model. The programming model of NSCALESPARK is similar to the NSCALE programming model wherein the user specifies the subgraphs of interest and the graph computation to be executed on them using the NSCALESPARK user API. The model has been further generalized to allow the user to specify a *query pattern* instead of a single *query vertex*. If there are no vertex or edge predicates, then an extracted subgraph is the union of k-hop neighborhoods around a matched pattern. This allows us to handle a larger variety of graph mining tasks [23]. Note that, any given vertex may participate as a matched vertex in a potentially large number of extracted subgraphs, making it even more important to carefully distribute and partition the graph as we do.

Graph Extraction and Packing (GEP) Module. In NSCALESPARK, the GEP module has been implemented on Apache Spark using a

series of RDD transformations in Scala. We describe the detailed steps below:

- Starting from a raw edge list representation of the underlying graph data, we use a series of coarse-grained transformations to incrementally construct and extract the relevant subgraphs of interest, instantiated in memory as a Spark RDD. Each subgraph is represented as a list of *vertices*, and information about edges is not maintained to keep the memory footprint low. As a result, we may extract supergraphs of the final subgraphs of interest.
- These subgraphs (as lists of vertices) are then provided as input to a shingle-based bin-packing algorithm [18]. The bin packing algorithm groups together subgraphs based on neighborhood similarity to minimize the memory footprint of the subgraphs when held in distributed memory. The final output of the bin packing algorithm is a vertex-to-partition mapping.
- Once this mapping information is obtained, it is then **joined** with the original graph data, to produce a memory efficient distributed instantiation of the extracted subgraphs.

Instantiating the subgraphs in distributed memory. We have built a graph library for NSCALESPARK that exposes a similar API and provides similar functionality as the graph library for NSCALE. The library provides the data structures for holding the subgraphs in memory as well as the bitmap implementations [18] required for the distributed and parallel execution of user computation. The graph library exports the popular BluePrints API, enabling the use of a large number of existing toolkits, applications and programs over large graphs. Once we made the BluePrints-based graph library available within the NSCALESPARK environment, we used the following steps to instantiate the subgraphs of interest in distributed memory.

- The subgraph structural information extracted was joined with the subgraph-to-partition information obtained from the bin packing algorithm within a coarse-grained map transformation. This enabled us to group the subgraphs using the partition number as the grouping key.
- The graph library API was used within the transformation to construct a graph object for each partition. Each graph object within a partition contains a set of subgraphs that had been binned together into the partition by the bin packing algorithm in the GEP phase. This ensures that all the subgraphs binned together are available in the memory of a single partition.
- The final output of this phase is a Spark RDD containing a set of BluePrint graph objects as described above, ready for executing user computation by the execution engine (Figure 5).

Executing user computation. The NSCALESPARK execution engine written in Java is primarily based on the NSCALE distributed execution engine [18] with some modifications. These modifications include design changes to the Master-worker architecture of the execution engine for enabling it to run within the Spark coarse-grained transformations. These transformations take the RDD graph objects as input for executing user computation and provide an output RDD to store the results of user computation. We explain the details of the execution phase below:

- We use the graph object Spark RDD obtained from GEP phase as input and apply a map transformation. The execution engine is instantiated within the map transformation creating a separate instance for each graph object within the RDD. *This design choice seamlessly enables us to use the Spark platform functionality to create an instantiation of a distributed execution engine for* NSCALESPARK.

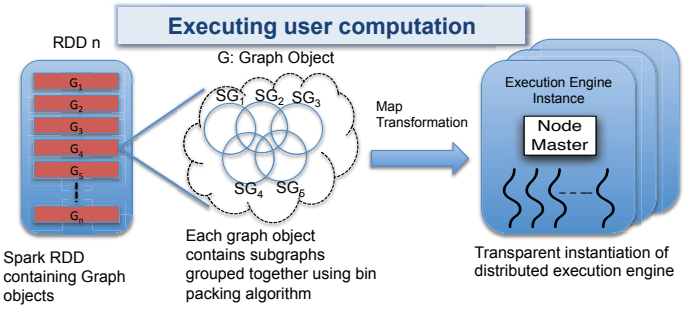

Figure 5: In-memory subgraph representation and execution of user computation.

- Within each instantiation of the execution engine instance, the Master process of the execution engine spawns several worker threads within a thread pool whose size is governed by the underlying hardware of the machine running the Spark executor instance.
- The worker threads execute the compute function written by the user (using the BluePrints API) on the subgraphs within each graph object of the RDD. The bitmap implementation provided by our library controls the scope of computation for each worker thread while enabling the parallel execution of user computation on subgraphs that have been stored in an overlapped fashion in memory.
- The design choice of using the bitmap-based NSCALE execution engine within the NSCALESPARK framework thus enables the distributed execution of user computation while minimizing memory consumption by exploiting overlap among the neighborhoods of interest.

The NSCALESPARK design draws from a unique performance advantage of the underlying Spark platform. Once an RDD is created, it can be persisted in memory and be repeatedly used for different analysis tasks. The NSCALESPARK design benefits from this wherein the graph RDD object created by the GEP phase can be persisted and used as input for several graph analytics tasks thus amortizing the cost of GEP phase across different analytics tasks. This also gives us the ability to meaningfully compose more complex tasks as chains wherein the output of a previous task can be directly fed as input to the next tasks in the chain.

4. EXPERIMENTAL EVALUATION

We performed an extensive evaluation of NSCALESPARK, and compared it to NSCALE (on Hadoop) and GraphX (on Apache Spark). We briefly discuss some additional implementations details of NSCALESPARK here, and describe the experimental setup.

Implementation Details. NSCALESPARK has been written in scala and deployed on an Apache Spark Cluster. The framework implements and exports the generic BluePrints API to write graph computations. The GEP module takes the subgraph extraction query, the bin packing heuristic to be used and the bin capacity. The Spark platform distributes the user computation and the execution engine library using the distributed cache mechanism to the appropriate machines on the cluster. The execution engine has been parametrized to vary its execution modes, and use different batch

Dataset	# Nodes	# Edges	Avg Degree	Avg Clust Coeff	# Triangles	Diameter
Notre Dame Web Graph	325729	2,994,268	9.19	0.2346	8910005	46
Google Web Graph	875713	10,210,078	11.66	0.5143	13391903	21
Wikipedia Talk Network	2,394,385	10,042,820	4.2	0.0526	9203519	9
LiveJournal Social Network	4,847,571	137,987,546	28.5	0.2741	285730264	16

Table 1: Dataset Statistics

sizes and bitmap construction techniques. In our current implementation of NSCALESPARK we focus on one-pass analytics to ascertain the feasibility and functionality of the port. We are currently working on adding support for iterative applications such as PageRank that NSCALE supports.

Data Sets. We conducted experiments using several different datasets, majority of which have been taken from the Stanford SNAP dataset repository [4] (see Table 1 for details and some statistics).

- **Web graphs:** We have used two different web graph datasets: *Notre Dame Web Graph*, and *Google Web Graph*; in all of these, the nodes represent web pages and directed edges represent hyperlinks between them.
- **Communication/Interaction networks:** We use the *Wikipedia Talk network*, created from the talk pages of registered users on Wikipedia until Jan 2008.
- **Social networks:** We also use one social network dataset: the *Live Journal social network*.

Graph Applications. We evaluate NSCALESPARK over 4 different applications. Two of them, namely, Local Clustering Coefficient (LCC), and Link Prediction using Personalized Page Rank (PPR), are described in Section 2. In addition, we used:

- **Triangle Counting (TC):** Here the goal is to count the number of triangles each vertex is part of. These statistics are very useful for complex network analysis [12] and real world applications such as spam detection, link recommendation, etc.
- **Motif Counting (MC):** Network motifs are subgraphs that appear in complex networks which have important applications in biological networks and other domains. Counting network motifs over large graphs involves identifying and counting specific subgraph patterns (e.g., Feed-forward loops) in the neighborhood of every query vertex that the user is interested in.

Comparison platforms. We compare NSCALESPARK with two different graph programming frameworks.

- **NScale.** NSCALE deployed on a Apache YARN framework.
- **GraphX [8].** GraphX is a graph programming library that sits on top of Apache Spark. We used the GraphX library version 2.10 over Spark version 1.4.0 with HDFS as the underlying storage layer.

Evaluation metrics. We use the following evaluation metrics to evaluate the performance of NSCALESPARK.

- **Computational Effort ($\mathcal{CE}$).** $\mathcal{CE}$ captures the total cost of doing analytics on a cluster of nodes deployed in the cloud. Let $T = \{T_1, T_2, ..., T_N\}$ be the set of tasks (or processes) deployed by the framework on the cluster during execution of the analytics task. Also, let t_i be the time taken by the task T_i to be executed on node i. We define $\mathcal{CE} = \sum_{i=1}^{N} t_i$. The metric captures the cost of doing data analytics in terms of *node-secs* which is appropriate for the cloud environment.
- **Execution Time.** This is the measure of the wall clock time or elapsed time for executing an end-to-end graph computation on a cluster of machines. It includes the time taken by the GEP phase for extracting the subgraphs as well as the time taken by the distributed execution engine to execute the user computation on all subgraphs of interest.
- **Cluster Memory.** Here we measure the maximum total physical memory used across all nodes in the cluster.

Experimental Setup. We used a 16 node cluster wherein each data node has 2 4-core Intel Xeon E5520 processors, 24GB RAM and 3 2TB disks. The cluster runs Apache YARN (MRv2 on Cloudera's CDH version 5.1.2), Apache Zookeeper for coordination and Apache Spark 1.4.0. Each process on this cluster runs in a container with a max memory capacity restricted to 15GB with a maximum of 6 processes per physical machine. We have evaluated our NSCALESPARK prototype on the Apache Spark platform with 15 executor instances and one driver instance with 15GB of memory available for each.

Experimental Results. Figure 6 shows the results for the performance comparisons of NSCALESPARK with NSCALE and GraphX. Comparing the performance of NSCALESPARK with NSCALE in terms of computational effort (CE–node secs) for two different data sets Web NotreDame and Web Google (Figures 6(a), 6(b)), we see that NSCALESPARK performs a little better which can be attributed to a better performance of the GEP phase on the Spark platform as compared to a multistage map-reduce implementation in Hadoop. As far as the memory consumption is concerned (Figures 6(c), 6(d)), NSCALESPARK consumes a little more memory than NSCALESPARK. The maximum virtual memory actually used by the Spark instance was 25.3GB for this set of experiments.

Next we compare the performance of NSCALESPARK with NSCALE and GraphX for (1) the LCC application, which requires computation over 1-hop neighborhoods around the query vertices and (2) the PPR application, which requires computation over 2-hop neighborhoods around the query vertices. We see that our implementation of NSCALESPARK performs much better than GraphX both in terms of $\mathcal{CE}$ and cluster memory for LCC. For PPR, GraphX did not finish for even the smallest of the datasets; specifically, many executor instances start thrashing and get lost/crash, and the scheduler tries to start the same tasks on other executor instances which also eventually run out of memory. This better performance can be attributed to a better design of the abstractions that hold the graph in much lesser memory and a better execution model which can take advantage of overlapped execution.

Finally Figure 7 shows the performance breakdown of the different components of NSCALESPARK in terms of the computational effort. We see that similar to NSCALE the user computation is still the major part of the computational effort as compared to the GEP phase. GEP phase in NSCALESPARK has been further broken down into subgraph extraction, bin packing and the actual construction of the graph RDD object.

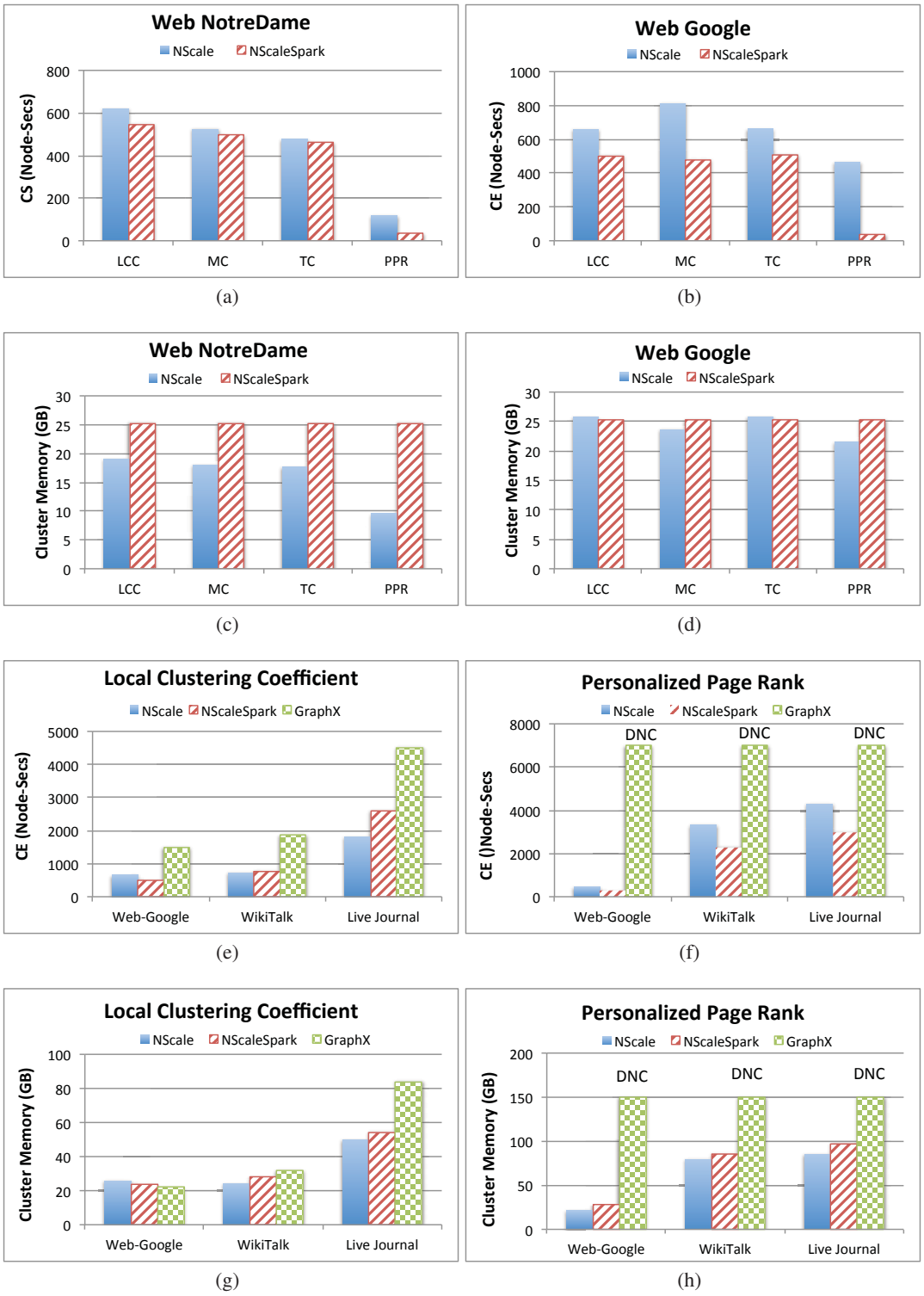

Figure 6: NSCALESPARK Performance : (a-b) Computational Effort (CE (node-secs)) comparison with NSCALE; (c-d) Cluster memory (GB) comparison with NSCALE; (e-h) Performance comparison with NSCALE and GraphX in terms of $\mathcal{CE}$ and Cluster memory. (DNC: Did not complete due to insufficient memory/thrashing.)

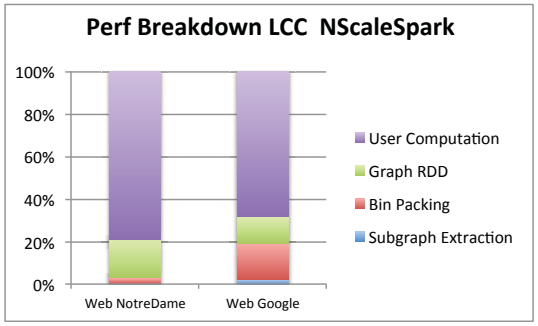

Figure 7: Performance breakdown of NSCALESPARK.

5. CONCLUSION

In this paper, we described the implementation of a subgraph-centric graph programming framework on Apache Spark, which can handle a large class of graph analysis tasks that are inefficient to execute using the popular vertex-centric programming framework. We argue that the subgraph-centric framework, where the users can write computations against entire subgraphs or multi-hop neighborhoods in the graph, is more natural both for ease-of-use and efficiency. Our comprehensive experimental evaluation illustrates the ability of our framework to execute a variety of graph analytics tasks on very large graphs with much better performance than GraphX, while consuming significantly fewer resources. We are currently working on enriching the programming model to support a large class of graph analysis and mining tasks through support for more general graph extraction queries; we are also working on developing better partitioning and placement algorithms for handling such analysis tasks. We are aiming to do an open-source release of the NSCALESPARK system in the near future.

Acknowledgments: This work was supported by NSF under grant IIS-1319432, and an IBM Collaborative Research Award.

6. REFERENCES

[1] BluePrints API: https://github.com/tinkerpop/blueprints/wiki.

[2] Furnace: https://github.com/tinkerpop/furnace/wiki.

[3] Gremlin: http://github.com/tinkerpop/gremlin/wiki.

[4] Stanford Network Analysis Project: https://snap.stanford.edu.

[5] L. Akoglu, M. McGlohon, and C. Faloutsos. OddBall: spotting anomalies in weighted graphs. In *PAKDD*, 2010.

[6] L Backstrom and J Leskovec. Supervised random walks: Predicting and recommending links in social networks. In *WSDM*, 2011.

[7] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A System for Searching the Social Graph. *Proc. VLDB Endow.*, 2013.

[8] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[9] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *WWW*, 2013.

[10] I. Hoque and I. Gupta. LFGraph: Simple and Fast Distributed Graph Analytics. In *TRIOS*, 2013.

[11] J. Huang., D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. In *PVLDB*, 2011.

[12] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics*, 2012.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB*, 2012.

[14] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD, 2010*.

[15] J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. In *NIPS*, 2012.

[16] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 2002.

[17] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

[18] A. Quamar, A. Deshpande, and J. Lin. NScale: Neighborhood-centric Large-Scale Graph Analytics in the Cloud. *VLDB J.*, 2015.

[19] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[20] J. Seo, S. Guo, and M. S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, 2013.

[21] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 2013.

[22] Y. Simmhan, A. G. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. S. Raghavendra, and V. K. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. *CoRR*, 2013.

[23] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, 2015.

[24] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB*, 2013.

[25] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*, 2013.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI'12.