# *GoFFish*: A Sub-graph Centric Framework for Large-Scale Graph Analytics

Yogesh Simmhan[1], Alok Kumbhare[2], Charith Wickramaarachchi[2], Soonil Nagarkar[2], Santosh Ravi[2], Cauligi Raghavendra[2], and Viktor Prasanna[2]

[1] Indian Institute of Science, Bangalore, India
[2] University of Southern California, Los Angeles, USA
simmhan@serc.iisc.in,
{kumbhare,cwickram,snagarka,sathyavi,raghu,prasanna}@usc.edu

**Abstract.** Vertex centric models for large scale graph processing are gaining traction due to their simple distributed programming abstraction. However, pure vertex centric algorithms under-perform due to large communication overheads and slow iterative convergence. We introduce *GoFFish* a scalable sub-graph centric framework co-designed with a distributed persistent graph storage for large scale graph analytics on commodity clusters, offering the added natural flexibility of shared memory sub-graph computation. We map Connected Components, SSSP and PageRank algorithms to this model and empirically analyze them for several real world graphs, demonstrating *orders of magnitude improvements*, in some cases, compared to Apache Giraph's vertex centric framework.

## 1 Introduction

One defining characteristic of complexity in "Big Data" is the intrinsic interconnectedness, endemic to novel applications in both the Internet of Things and Social Networks. Such graph datasets offer unique challenges to scalable analysis, even as they are becoming pervasive. There has been significant work on parallel algorithms and frameworks for large graph applications on HPC clusters [1][1], massively multi-threaded architectures [2], and GPUs [3]. Our focus here, however, is on leveraging *commodity hardware* for scaling graph analytics. Such distributed infrastructure, including Clouds, have democratized resource access, as evidenced by popular programming frameworks like MapReduce. While MapReduce's tuple-based approach is ill-suited for many graph applications [4], recent *vertex-centric programming abstractions* [5,6], like Google's Pregel and its open-source version, Apache Giraph [7], marry the ease of specifying a uniform logic for each vertex with a *Bulk Synchronous Parallel (BSP)* execution model to scale[2]. Independent vertex executions in a distributed environment are interleaved with synchronized message exchanges across them to form iterative *supersteps*.

However, there are short-comings to this approach. (1) Defining an individual vertex's logic forces costly messaging even within vertices in one partition. (2) Porting shared memory graph algorithms to efficient vertex centric ones can be non-trivial. (3)

[1] The Graph 500 List, http://www.graph500.org
[2] Scaling Apache Giraph to a Trillion Edges, *Facebook Engineering*, http://on.fb.me/1czMarU

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 451–462, 2014.
© Springer International Publishing Switzerland 2014

The programming abstraction is decoupled from the data layout on disk, causing I/O penalties at initialization and runtime. A recent work [6] identified the opportunity of leveraging shared memory algorithms within a *partition*, but relaxed the programming model to operate on a whole partition, without guarantees of *sub-graph connectivity* within a partition or implicit use of *concurrency* across sub-graphs.

In this paper, (1) we propose a sub-graph centric programming abstraction, *Gopher*, for performing distributed graph analytics. This balances the flexibility of reusing shared-memory graph algorithms over *connected* sub-graphs while leveraging sub-graph concurrency within a machine, and distributed scaling using BSP. (2) We couple this abstraction with an efficient distributed storage, *GoFS*. GoFS stores partitioned graphs across distributed hosts while coalescing connected sub-graphs within partitions, to optimize sub-graph centric access patterns. Gopher and GoFS are co-designed as part of the *GoFFish* graph analytics framework, and are empirically shown here to scale for common graph algorithms on real world graphs, in comparison with Apache Giraph.

## 2 Background and Related Work

There is a large body of work on parallel graph processing [8] for HPC clusters [1], massively multi-threaded architectures [2], and GPGPUs [3], often tuned for specific algorithms and architectures [9]. For e.g., the STAPL Parallel Graph Library (SGL) [10] offers diverse graph data abstractions, and can express level-synchronous vertex-centric BSP and coarse-grained algorithms over sub-graphs, but not necessarily a sub-graph centric BSP pattern as we propose. SGL uses STAPL for parallel execution using OpenMP and MPI, which scales on HPC hardware but not on commodity clusters with punitive network latencies. Frameworks for commodity hardware trade performance in favor of scalability and accessibility – ease of use, resource access, and programming.

The popularity of MapReduce for large scale data analysis has extended to graph data as well, with research techniques to scale it to peta-byte graphs for some algorithms [11]. But the tuple-based approach of MapReduce makes it unnatural for graph algorithms, often requiring new programming constructs [12], platform tuning [4], and repeated reads/writes of the entire graph to disk. Google's recent Pregel [5] model uses iterative supersteps of vertex centric logic executed in a BSP model [13]. Here, users implement a `Compute` method for a single vertex, with access to its value(s) and outgoing edge list. `Compute` is executed concurrently for each vertex and can emit messages to neighboring (or discovered) vertices. Generated messages are delivered in bulk and available to the target vertices only after all `Computes` complete, forming one barriered *superstep*. Iterative level-synchronized supersteps interleave computation with message passing. The vertices can `VoteToHalt` in their `Compute` method at any superstep; any vertex that has voted to halt is not invoked in the next superstep unless it has input messages. The application terminates when all vertices vote to halt and there are no new input messages available. Pseudocode to find the maximal vertex using this model is shown in Alg. 1.. Apache Giraph [7] is an open source implementation of Google's Pregel, and alternatives such as Hama [14], Pregel.NET [15] and GPS [16] also exist.

Despite the programming elegance, there are key scalability bottlenecks in Pregel: (1) the number of messages exchanged between vertices, and (2) the number of synchronized supersteps required for completion. Message passing is done either in-memory

(for co-located vertices) or over the network, while barrier synchronization across distributed vertices is centrally coordinated. This often makes them communication bound on commodity hardware, despite use of `Combiners` [5] for message aggregation. Secondly, the default hashing of vertices to machines exacerbates this though better partitioning shows mixed results [16,6]. Even in-memory message passing causes memory pressure [15]. For e.g., GPS [16] performs dynamic partition balancing and replication on Pregel and our prior work [15] used a swathe-based scheduling to amortize messaging overhead. But these engineering solutions do not address key limitations of the abstraction, which also leads to large number of supersteps to converge for vertex centric algorithms (e.g.∼30 for PageRank, or graph diameter for Max Vertex).

Recently, Tian, et al. [6] recognized these limitations and propose a *partition centric* variant, *Giraph++*. Here, users' `Compute` method has access to all vertices and edges in a partition on a machine, and can define algorithms that operate on a whole partition within a superstep before passing messages from *boundary vertices* of the partition to neighboring vertices. They also partition the graph to minimize average *ncuts*. Such local compute on the coarse partition can reduce the number of messages exchanged and supersteps taken, just as for us. But this approach falls short on several counts. (1) Though the partitions are called "sub-graphs" in Giraph++, these sub-graphs are not connected components. This limits the use of shared-memory graph algorithms that operate on connected graphs, and can lead to a suboptimal algorithm operating collectively on hundreds of sub-graphs in a partition. This also puts the onus on the user to leverage concurrency across sub-graphs in a partition. Our proposed abstraction and execution model *a priori* identifies *sub-graphs as locally connected components*; users define their `Compute` method on these. Our engine also automatically executes sub-graphs in parallel on the local machine. (2) Their `Compute` can send messages to only boundary vertices. We also allow messages to be sent to sub-graphs, fully exploiting the abstraction. (3) Our distributed graph storage is designed for sub-graph access patterns to offer data loading efficiencies, and extends beyond just prior graph partitioning.

**Algorithm 1.** Max Vertex Value using Vertex Centric Model

```
 1: procedure COMPUTE(Vertex myVertex, Iterator⟨Message⟩ M)
 2:     hasChanged = (superstep == 1) ? true : false
 3:     while M.hasNext do                ▷ Update to max message value
 4:         Message m ← M.next
 5:         if m.value > myVertex.value then
 6:             myVertex.value ← m.value
 7:             hasChanged = true
 8:     if hasChanged then                ▷ Send message to neighbors
 9:         SENDTOALLNEIGHBORS(myVertex.value)
10:     else
11:         VOTETOHALT()
```

Distributed GraphLab [17] is another popular graph programming abstraction, optimized for local dependencies observed in data mining algorithms. GraphLab too uses an iterative computing model based on vertex-centric logic, but allows asynchronous execution with access to vertex neighbors. Unlike Pregel, it does not support graph mutations. There are other distributed graph processing systems such as Trinity [18]

that offer a shared memory abstraction in a distributed memory infrastructure. Here, algorithms use both message passing and a distributed address space called *memory cloud*. However, this assumes large memory machines with high speed interconnects. We focus on commodity cluster and do not make such hardware assumptions.

## 3 Sub-graph Centric Programming Abstraction

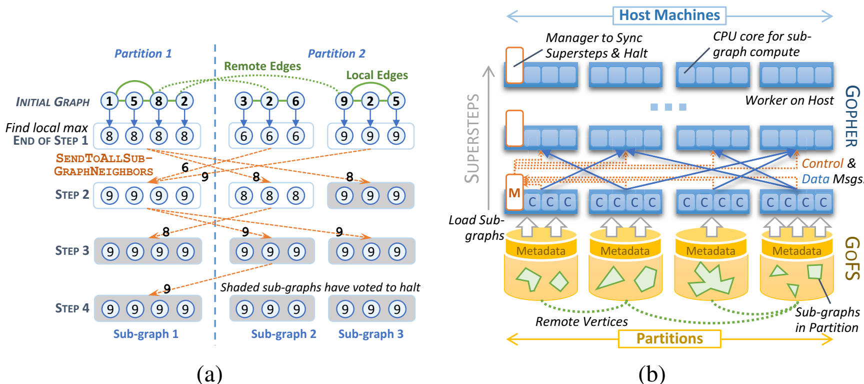
We propose a sub-graph centric programming abstraction that targets the deficiencies of a vertex centric BSP model. We operate in a distributed environment where the graph is $k - way$ partitioned over its vertices across $k$ machines. We define a **sub-graph** as a connected component within a partition of an undirected graph; they are weakly connected if the graph is directed. The Fig. 1(a) shows two partitions with three sub-graphs. Two sub-graphs do not share the same vertex but can have *remote edges* that connect their vertices (dotted edges in Fig. 1(a)), as long as the sub-graphs are on different partitions. If two sub-graphs on the same partition share a *local edge* (solid edges), by definition they are merged into a single sub-graph. A partition can have one or more sub-graphs and the set of all sub-graphs forms the complete graph. Specific partitioning approaches are discussed later, and each machine holds one partition. Sub-graphs behave like "meta vertices" with remote edges connecting them across partitions.

Formally, let $P_i = \{\mathbb{V}_i, \mathbb{E}_i\}$ be a graph partition $i$ where $V_i$ and $E_i$ are the set of vertices and edges in the partition. We define a *sub-graph* S in $P_i$ as $S = \{V, E, R | v \in V \Rightarrow v \in \mathbb{V}_i; e \in E \Rightarrow e \in \mathbb{E}_i; r \in R \Rightarrow r \notin \mathbb{V}_i; \forall u, v \in V \exists$ an undirected path between $u, v$; and $\forall r \in R\ \exists v \in V$ s.t. $e = \langle v, r \rangle \in E\}$ where V is a set of local vertices, E is a set of edges and R is a set of remote vertices.

Each sub-graph is treated as an independent unit of computation within a BSP superstep. Users implement the following method signature:

```
Compute(Subgraph, Iterator<Message>)
```

The `Compute` method can access the sub-graph topology and values of the vertices and edges. The values are mutable though the topology is constant. This allows us to fully traverse the sub-graph up to the boundary remote edges *in-memory, within a single*




(a) (b)

**Fig. 1.** (a) Sub-graph centric Max Vertex using Alg. 2.. Dashed arrows show messages passed. (b) Sub-graph centric data access from GoFS and BSP execution by Gopher.

*superstep* and accumulate values of the sub-graph or update values for the local vertices and edges. Different sub-graphs communicate by message passing, with messages exchanged at synchronized superstep boundaries in a BSP model. Several methods enable this messaging. Algorithms often start by sending messages to neighboring sub-graphs.

```
SendToAllSubGraphNeighbors(Message)
```

As other sub-graphs are discovered across supersteps, two other methods are useful:

```
SendToSubGraph(SubGraphID, Message)
SendToSubGraphVertex(SubGraphID, VertexID, Message)
```

We allow a (costly) broadcast to all sub-graphs, though it should be used sparingly.

```
SendToAllSubGraphs(Message)
```

As with Pregel, the `Compute` method can invoke `VoteToHalt`. The application terminates when all sub-graphs have halted and there are no new input messages.

Alg. 2. presents the sub-graph centric version for finding the maximum vertex value in a graph. Fig. 1(a) illustrates its execution. The `Compute` method operates on a sub-graph *mySG*. Lines 2–6 are executed only for the first superstep, where each sub-graph's value is initialized to largest of its vertices. Subsequently, the algorithm is similar to the vertex centric version: we send the sub-graph's value to its neighboring sub-graphs and update the sub-graph's value to the highest value received, halting when there is no further change. At the end, each sub-graph has the value of the largest vertex.

**Algorithm 2.** Max Vertex using Sub-Graph Centric Model

```
 1: procedure COMPUTE(SubGraph mySG, Iterator⟨Message⟩ M)
 2:     if superstep = 1 then                ▷ Find local max in subgraph
 3:         mySG.value ← −∞
 4:         for all Vertex myVertex in mySG.vertices do
 5:             if mySG.value < myVertex.value then
 6:                 mySG.value ← myVertex.value
 7:     hasChanged = (superstep == 1) ? true : false
 8:     while M.hasNext do
 9:         Message m ← M.next
10:         if m.value > mySG.value then
11:             mySG.value ← m.value
12:             hasChanged = true
13:     if hasChanged then
14:         SENDTOALLSUBGRAPHNEIGHBORS(mySG.value)
15:     else
16:         VOTETOHALT()
```

Compared to the vertex centric algorithm, the sub-graph centric version reduces the number of supersteps taken since the largest value discovered at any superstep propagates through the entire sub-graph in the same superstep. For e.g., for the graph in Fig. 1(a), the vertex centric approach takes 7 supersteps while we use 4. Also, this reduces the cumulative number of messages exchanged on the network. In the worst case, when a sub-graph is trivial (has one vertex), we degenerate to a vertex centric model.

**Benefits.** Our elegant extension of the vertex centric model offers three key benefits.

*1) Messages Exchanged.* Access to the entire sub-graph enables the application to make a significant progress within each superstep while reducing costly message exchanges that cause network overhead, disk buffering or memory pressure, between supersteps. While Pregel allows `Combiners` per worker, they operate after messages are generated while we preclude message generation. Giraph++ has similar advantages, but no better; as sub-graphs in a partition are disconnected, they do not exchange messages.

*2) Number of Supersteps.* Depending on the algorithm, a sub-graph centric model can reduce the number of supersteps required compared to Pregel, thereby limiting synchronization overheads. Also, the time taken by a superstep is based on its slowest sub-graph, with a wide distribution seen across sub-graphs [15]. Reducing the supersteps mitigates this skew. For traversal algorithms, the number of supersteps is a function of the graph *diameter*. Using a sub-graph centric model, this reduces to the diameter of the meta-graph where the sub-graphs form meta-vertices. In the best case (a linear chain), the number of supersteps can reduce proportional to the number vertices in a sub-graph, while for a trivial sub-graph, it is no worse. These benefits translate to Giraph++ too.

*3) Reuse of Single-machine Algorithms.* Lastly, our approach allows direct reuse of efficient shared-memory graph algorithms on a sub-graph, while using a BSP model across supersteps. The change from a simple vertex-centric approach is incremental, but the performance improvement stark. e.g. In Alg. 2., but for the shaded lines 2–6 which operates on the whole sub-graph, other lines are similar to Alg. 1.. This has two benefits relative to Pregel and Giraph++: (1) We can leverage optimal single machines algorithms for sub-graphs, even leveraging GPGPU accelerators, with the added guarantee that the *sub-graphs are connected* (unlike Giraph++). This ensures traversals reach all sub-graph vertices and avoids testing every vertex in the partition independently. Second, when a partition has multiple sub-graphs, as is often, we can exploit concurrency across them automatically. While the degree of parallelism is not as high as vertex centric (Pregel), it is better than treating the partition as one computation unit (Giraph++).

## 4 Architecture

*GoFFish* is a scalable software framework for storing graphs, and composing and executing graph analytics[3]. A *Graph oriented File System (GoFS)* and *Gopher execution engine* are co-designed *ab initio* to ensure efficient distributed storage for sub-graph centric data access patterns. The design choices target commodity or virtualized hardware with Ethernet and spinning disks. GoFFish is implemented in Java.

**GoFS Distributed Graph Store.** GoFS is a distributed store for partitioning, storing and accessing graph datasets across hosts in a cluster. Graphs can have both a *topology* and *attributes* associated with each vertex and edge. The former is an adjacency list of uniquely labeled vertices and (directed or undirected) edges connecting them. Attributes are a list of name-value pairs with a schema provided for typing. Input graphs are partitioned across hosts, one partition per machine, using the METIS tool [19] to balance vertices per partition and minimize edge cuts (Fig. 1(b)).

GoFS uses a sub-graph oriented model for mapping the partition's content to *slice files*, which form units of storage on the local file system. We identify all sub-graphs

[3] `https://github.com/usc-cloud/goffish`

in the partition – components that are (weakly) connected through local edges, and a partition with $n$ vertices can have between 1 to $n$ sub-graphs. Each sub-graph maps to one *topology slice* that contains local vertices, local edges and remote edges, with references to partitions holding the destination remote vertex, and several *attribute slices* that hold their names and values. We use *Kryo*[4] for compact object storage on disk.

GoFS is a *write once-read many* scalable data store rather than a database with rich query support. The GoFS Java API allows clients to access a graph's metadata, attribute schema and sub-graphs present in the local partition. Specific sub-graphs and select attributes can be loaded into memory and traversed. Remote edges in a sub-graph resolve to a remote partition/sub-graph/vertex ID that can be used to send messages to.

**Gopher Sub-graph Centric Framework.** The Gopher programming framework implements our proposed sub-graph centric abstractions, and executes them using the *Floe* [20] dataflow engine on a Cloud or cluster in conjunction with GoFS. Users implement their algorithm in Java within a `Compute` method where they get access to a local sub-graph object and data messages from the previous superstep. They use `Send*` methods to send message to the remote sub-graphs in the next superstep and can `VoteToHalt()`. The same `Compute` logic is executed on every sub-graph in the graph, for each superstep.

The Gopher framework has a *compute worker* running on each machine and a *manager* on one machine. The workers initially load all local sub-graphs for that graph into memory from GoFS. For every superstep, the worker uses a multi-core-optimized task pool to invoke the `Compute` on each sub-graph, transparently leveraging concurrency within a partition. `Send*` messages are resolved by GoFS to a remote partition and host. The worker aggregates messages destined for the same host and sends them asynchronously to the remote worker while the compute progresses.

Once the `Compute` for all sub-graphs in a partition complete, the worker flushes pending messages to remote workers and signals the manager. Once the manager receives signals from all workers, it broadcasts a *resume* signal to the workers to start their next superstep and operate on input messages from the previous superstep. `Compute` is stateful for each sub-graph; so local variables are retained across supersteps. When a worker does not have to call `Compute` for any of its sub-graphs in a superstep, because all voted to halt *and* have no input messages, it sends a *ready to halt* signal to the manager. When all workers are ready to halt, the manager terminates the application.

**Storage-Compute Co-design.** Co-designing data layout and execution models is beneficial, as seen with Hadoop and HDFS. GoFFish uses sub-graphs as a logical unit of storage and computation; hence our data store first partitions the graph followed by sub-graph discovery. Partitioning minimizes network costs when loading sub-graphs into Gopher. We use existing partitioning tools (METIS) to balance vertices and minimize edge cuts. Ideally, we should also balance the number of sub-graphs per partition and ensure uniform size to reduce compute skew in a superstep. Further, having multiple sub-graphs in a partition can leverage the concurrency across sub-graphs. Such schemes are for future work. We also balance the disk latency against bytes read by slicing sub-graphs into topology and attributes files. For e.g. a graph with 10 edge attributes that uses only the *weight* attribute in an algorithm needs to load only one attribute slice.

[4] Kryo serialization framework, `https://github.com/EsotericSoftware/kryo`

## 5 Evaluation of Sub-graph Centric Algorithms on GoFFish

We present and evaluate several sub-graph centric versions of common graph algorithms, both to illustrate the utility of our abstraction and the performance of GoFFish. We comparatively evaluate against Apache Giraph, a popular implementation of Pregel's vertex centric model, that uses HDFS. We use the latest development version of Giraph, at the time of writing, which includes recent performance enhancements. Sub-graph centric Gopher and vertex centric Giraph algorithms are implemented for: *Connected Components*, *Single Source Shortest Path (SSSP)* and *PageRank*.

**Experimental Setup and Datasets.** We run these experiments on a modest cluster of 12 nodes, each with an 8-core Intel Xeon CPU, 16 GB RAM, 1 TB SATA HDD, and connected by Gigabit Ethernet. This is representative of commodity clusters or Cloud VMs accessible to the long tail of science rather than HPC users. Both Giraph and GoFFish are deployed on all nodes, and use Java 7 JRE for 64 bit Ubuntu Linux. The GoFFish manager runs on one node.

We choose diverse real world graphs (Table 1): California road network (**RN**), Internet topology from traceroute statistics (**TR**)), and LiveJournal social network (**LJ**). RN is a small, sparse network with a small and even edge degree distribution, and a large diameter. LJ is dense, with powerlaw edge degrees and a small diameter. TR has a powerlaw edge degree, with a few highly connected vertices. Unless otherwise stated, we report average values over three runs each for each experiment.

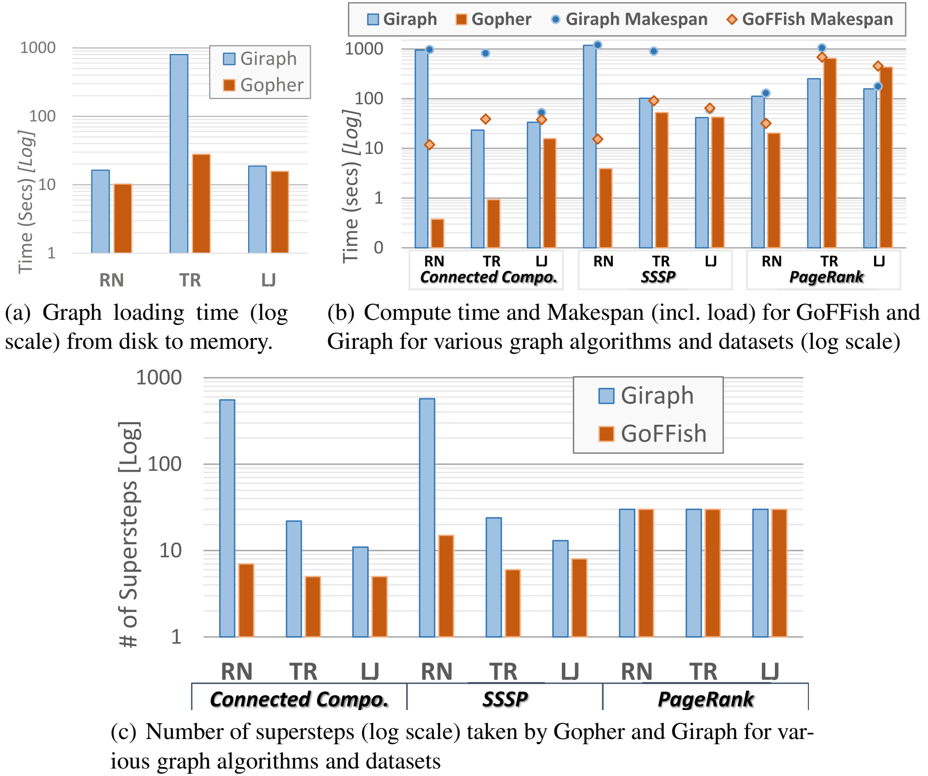
**Table 1.** Characteristics of graph datasets used in evaluation

| **Graph** | *Vertices* | *Edges* | *Diameter* | *WCC* |
|---|---|---|---|---|
| **RN** | 1,965,206 | 2,766,607 | 849 | 2,638 |
| **TR** | 19,442,778 | 22,782,842 | 25 | 1 |
| **LJ** | 4,847,571 | 68,475,391 | 10 | 1,877 |

**Summary Results.** We compare the end-to-end time (makespan) for executing an algorithm on GoFFish and on Giraph. This includes two key components: the time to load the data from storage, which shows the benefits of sub-graph oriented distributed storage, and the time to run the sub-graph/vertex centric computation. which shows relative benefits of the abstractions. Fig. 2(a) highlights the data loading time per graph on both platforms; this does not change across algorithms. Fig. 2(b) give the execution time as a *bar-plot* for various algorithms and datasets once data is loaded, as well as the makespan that includes the compute and load time, as a *dot-plot*. Also shown in Fig. 2(c) is the number of supersteps taken to complete the algorithm for each combination.

One key observation is that GoFFish's makespan is smaller than Giraph for all combinations but two, PageRank and SSSP on LJ. The performance advantage ranges from $81\times$ faster for Connected Components on RN to $11\%$ faster for PageRank on TR. These result from abstraction, design and layout choices, as we discuss. In some, Giraph's data loading time from HDFS dominates (TR), in others, Gopher's algorithmic advantage significantly reduces the number of supersteps (RN for SSSP), while for a few, Gopher's compute time over sub-graphs dominates (PageRank on LJ).

**Connected Components.** Connected components identify maximally connected sub-graphs within an undirected graph such that there is path from every vertex to every




(a) Graph loading time (log scale) from disk to memory.

(b) Compute time and Makespan (incl. load) for GoFFish and Giraph for various graph algorithms and datasets (log scale)

(c) Number of supersteps (log scale) taken by Gopher and Giraph for various graph algorithms and datasets

**Fig. 2.** Comparison of GoFFish and Giraph for all Graph Algorithms and Datasets

other vertex in the sub-graph. The sub-graph and vertex centric algorithms are similar to the Maximum Vertex Value algorithm [21]. In effect, we perform a *breadth first traversal* rooted at the sub-graph with the largest vertex ID, with each superstep propagating the value one level deeper till the farthest connected sub-graph is reached. Finally, all vertices are labeled with the component ID (i.e. largest vertex ID) they belong to.

The computational complexity of this algorithm is $O((d+1) \times v/p)$, where $d$ is the diameter of the graph (specifically, of the largest connected component) constructed by treating each sub-graph as a meta vertex, $v$ is the number of vertices in the graph and $p$ is the number of machines (partitions). The key algorithmic optimization here comes from reducing the number of supersteps $(d+1)$ relative to the vertex centric model.

As a result, Connected Components for GoFFish performs significantly better than Giraph for all three data sets – from $1.4\times$ to $81\times$. Fig. 2(c) shows the number of supersteps is much smaller for Gopher compared to Giraph, taking between 5 (TR, LJ) and 7 (RN) supersteps for Connected Components while Giraph takes between 11 (LJ) and 554 (RN). The superstep time in itself is dominated by the synchronization overhead. The ratio of compute times improvements between Giraph and Gopher is highly correlated ($R^2 = 0.9999$) with the vertex-based diameter of the graph (Table 1), *i.e., larger the vertex-based graph diameter, greater the opportunity to reduce sub-graph-based diameter, lesser the number of supersteps, and better that Gopher algorithm performs.*

Gopher's makespan for TR graph is $21\times$ better than Giraph due to much faster data loading by GoFS ($27secs$ vs. $798secs$). Giraph's HDFS, which balances the vertices across partitions, has to move one vertex with 1M edges that takes punitively long.

**Algorithm 3.** Sub-Graph Centric Single Source Shortest Path

```
 1: procedure COMPUTE(SubGraph mySG, Iterator⟨Message⟩ M)
 2:     openset ← ∅                  ▷ Vertices with improved distances
 3:     if superstep = 1 then        ▷ Initialize distances
 4:         for all Vertex v in mySG.vertices do
 5:             if v = SOURCE then
 6:                 v.value ← 0          ▷ Set distance to source as 0
 7:                 openset.add(v)       ▷ Distance has improved
 8:             else
 9:                 v.value ← −∞         ▷ Not source vertex
10:     for all Message m in M do        ▷ Process input messages
11:         if mySG.vertices[m.vertex].value > m.value then
12:             mySG.vertices[m.vertex].value ← m.value
13:             openset.add(m.vertex)    ▷ Distance improved
14:             ▷ Call Dijkstras and get remote vertices to send updates
15:     remoteSet ← DIJKSTRAS(mySG, openset)
16:             ▷ Send new distances to remote sub-graphs/vertices
17:     for all ⟨remoteSG,vertex,value⟩ in remoteSet do
18:         SENDTOSUBGRAPHVERTEX(remoteSG, vertex, value)
19:     VOTETOHALT( )
```

**SSSP.** Intuitively, the sub-graph centric algorithm for Single Source Shortest Path (SSSP) finds the shortest distances from the source vertex to all internal vertices (i.e. not having a remote edge) in the sub-graph holding the source in one superstep using DIJKSTRAS (Alg. 3.). It then sends the updated distances from the vertices having a remote edge to their neighboring sub-graphs. These sub-graphs propagate the changes internally in one superstep, and to their neighbors across supersteps, till the distances quiesce.

DIJKSTRAS has a compute complexity of $O((e \cdot log(v)))$ per superstep, where $e$ and $v$ are typically dominated by the largest active sub-graph. The number of supersteps is a function of the graph diameter $d$ measured through sub-graphs, and this takes $O(d)$ supersteps. For a partitions with large number of small sub-graphs, we can exploit concurrency across $c$ cores on that machine. While the time complexity per superstep is relatively larger for DIJKSTRAS, we may significantly reduce the number of supersteps taken for the algorithm to converge.

SSSP's compute time for GoFFish out-performs Giraph by $300\times$ and $2\times$ for RN and TR, respectively, while it is the same for LJ. Gopher takes reduced supersteps on RN and TR for SSSP, that is able to offset its higher computational complexity per superstep. But this complexity impacts LJ which has high edge density, while its small world network diameter does not reduce the number of supersteps. Hence SSSP for Gopher only matches, rather than outperforms, Giraph's compute time for LJ.

**PageRank.** For each superstep in a vertex centric PageRank [5], a vertex adds all input message values into $sum$, computes $0.15/v + 0.85 \times sum$ as its new value, and sends $value/g$ to its $g$ neighbors. The $value$ is $1/v$ initially, for $v$ vertices in the graph. An equivalent sub-graph centric approach does not confer algorithmic benefits; it takes

the same $\sim$30 supersteps to converge and each vertex operates independently in lock step, with an $O(30 \cdot \frac{v}{p \cdot c} \cdot g)$, for an average edge degree of $g$.

As shown in Fig. 2(b), Gopher under performs Giraph for PageRank for TR and LJ by $2.6\times$. It is $5.5\times$ faster for RN. TR's makespan offsets compute slowdown with data loading benefits. As observed, the fixed supersteps used for PageRank negates algorithmic benefits and the computation complexity per superstep for sub-graph centric is higher than for vertex centric. This also exacerbates the time skew across sub-graphs in a partition. For e.g., in LJ, many of the partitions complete their superstep within a range of $23 - 26secs$, but these are bound by single large sub-graphs in each partition which are stragglers, and cause $75\%$ of the cores to be idle. Giraph, on the other hand, has uniform vertex distribution across machines and each worker takes almost the same time to complete a superstep while fully exploiting fine grained vertex level parallelism. This highlights the deficiencies of the default partitioning model used by GoFS that reduces edge cuts and balances the number of vertices per machine, *without considering the number of sub-graphs that are present per partition, and their sizes*.

## 6 Discussion and Conclusions

We introduce a sub-graph centric programming abstraction for large scale graph analytics on distributed systems. This model combines the scalability of vertex centric programming with the flexibility of using shared-memory algorithms at the sub-graph level. The connected nature of our sub-graphs provides stronger guarantees for such algorithms and allows us to exploit degrees of parallelism across sub-graphs in a partition. The GoFFish framework offers Gopher, a distributed execution runtime for this abstraction, co-designed with GoFS, a distributed sub-graph aware storage that pre-partitions and stores graphs for data-local execution.

The relative algorithmic benefits of using a sub-graph centric abstraction can be characterized based on the class of graph algorithm and graph. For algorithms that perform full graph traversals, like SSSP, BFS and Betweenness Centrality, we reduce the number of supersteps to a function of the diameter of the graph based on sub-graphs rather than vertices. This can offer significant reduction. However, for powerlaw graphs that start with a small vertex based diameter, these benefits are muted.

The time complexity per superstep can be larger since we often run the single-machine graph algorithm on each sub-graph. The number of vertices and edges in large sub-graph will impact this. If there are many small sub-graphs in a partition, the number of sub-graphs becomes the limiting factor as we approach a vertex centric behavior, but this also exploits multi-core parallelism. For graphs with high edge density, algorithms that are a linear (or worse) function of the number of edges can take longer supersteps.

We empirically showed that GoFFish performs significantly better than Apache Giraph. These performance gains are due to both the partitioned graph storage and sub-graph based retrieval from GoFS, and a significant reduction in the number of supersteps that helps us complete faster. This offers a high compute to communication ratio.

We do recognize some short comings, with further research opportunities. Sub-graph centric algorithms are vulnerable to imbalances in number of sub-graphs per partition and non-uniformity in their sizes. This causes stragglers. Better partitioning to balance the sub-graphs can help. The framework is also susceptible to small-world graphs with

high edge degrees that have high sub-graph level computational complexity. Our software prototype offers opportunities for design and engineering optimizations.

## References

1. Gregor, D., Lumsdaine, A.: The Parallel BGL: A Generic Library for Distributed Graph Computations. In: Parallel Object-Oriented Scientific Computing, POOSC (2005)
2. Ediger, D., Bader, D.: Investigating Graph Algorithms in the BSP Model on the Cray XMT. In: Workshop on Multithreaded Architectures and Applications, MTAAP (2013)
3. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the gpu using cuda. In: IEEE High Performance Computing, HiPC (2007)
4. Lin, J., Schatz, M.: Design patterns for efficient graph algorithms in MapReduce. In: Workshop on Mining and Learning with Graphs, pp. 78–85. ACM (2010)
5. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: ACM International Conference on the Management of Data (SIGMOD), pp. 135–146. ACM (2010)
6. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From "Think Like a Vertex" to "Think Like a Graph". Proc. of the VLDB (PVLDB) 7(3), 193–204 (2013)
7. Avery, C.: Giraph: Large-scale graph processing infrastructure on hadoop. In: Hadoop Summit (2011)
8. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.: Challenges in parallel graph processing. Parallel Processing Letters 17(01), 5–20 (2007)
9. Buluç, A., Madduri, K.: Parallel breadth-first search on distributed memory systems. In: IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC). ACM (2011)
10. Harshvardhan, Fidel, A., Amato, N.M., Rauchwerger, L.: The STAPL Parallel Graph Library. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 46–60. Springer, Heidelberg (2013)
11. Papadimitriou, S., Sun, J.: DisCo: Distributed Co-clustering with Map-Reduce. In: IEEE International Conference on Data Mining, ICDM (2008)
12. Chen, R., Weng, X., He, B., Yang, M.: Large graph processing in the cloud. In: ACM International Conference on the Management of Data (SIGMOD), pp. 1123–1126. ACM (2010)
13. Gerbessiotis, A.V., Valiant, L.G.: Direct bulk-synchronous parallel algorithms. Journal of Parallel and Distributed Computing (JPDC) 22(2), 251–267 (1994)
14. Seo, S., Yoon, E.J., Kim, J., Jin, S., Kim, J.S., Maeng, S.: Hama: An efficient matrix computation with the mapreduce framework. In: IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE (2010)
15. Redekopp, M., Simmhan, Y., Prasanna, V.: Optimizations and analysis of bsp graph processing models on public clouds. In: IEEE Intl. Parallel & Distr. Proc. Symp., IPDPS (2013)
16. Salihoglu, S., Widom, J.: GPS: A Graph Processing System. In: International Conference on Scientific and Statistical Database Management, SSDBM (2013)
17. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. VLDB 5(8), 716–727 (2012)
18. Shao, B., Wang, H., Li, Y.: Trinity: A distributed graph engine on a memory cloud. In: ACM International Conference on the Management of Data, SIGMOD (2013)
19. Karypis, G., Kumar, V.: Analysis of multilevel graph partitioning. In: IEEE/ACM Intl. Conf. for High Performance Computing, Networking, Storage and Analysis, SC (1995)
20. Simmhan, Y., Kumbhare, A., Wickramachari, C.: Floe: A dynamic, continusous dataflow framework for elastic clouds. Technical report, USC (2013)
21. Kang, U., Tsourakakis, C.E., Faloutsos, C.: Pegasus: A peta-scale graph mining system implementation and observations. In: IEEE Intl. Conf. on Data Mining, ICDM (2009)