PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

Authors: Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin

Presented By: Rahul Yesantharao

03/31/2020

Paper Outline

- Presents the new PowerGraph framework for efficient Graph-Parallel processing.
- Presents a new abstraction for general graph-parallel algorithm design
- Focuses on designing a new system that behaves well on natural graphs
- Presents several key innovations that achieve better performance than existing graph-parallel frameworks, exemplified by Pregel and GraphLab

Context: Graph-Parallel Processing

- Graph-parallel processing is very important in the age of big data
- The term refers to algorithms that perform computations through "vertex-programs" that run on individual vertices and interact by sending data over edges.
- Many major machine learning and data mining (MLDM) algorithms, such as PageRank, Single-Source Shortest Path, and graph-coloring, can be encoded in this paradigm
- Existing frameworks, however, parallelize by using individual vertices as the unit of computation, and thus depend on vertices having low degrees to achieve good parallelism

Context: Natural Graphs

- Most real-world applications of graph-parallel frameworks are run over so-called “natural graphs”
- This term refers to graphs that represent data from the real world, such as social networks or the Internet
- However, these graphs have been shown to follow a power-law degree distribution, which implies that a small subset of the vertices have very high degrees
- This innate imbalance in natural graphs poses an issue for existing graph-parallel frameworks, which factor computation over vertices and thus depend on degree distribution for performance
- As α gets larger, the skewedness grows.

$$\mathbf{P}(d) \propto d^{-\alpha}$$

GAS Model

- The paper introduces a characterization of the general graph-parallel problem
- The inputs are a sparse graph {V, E} and a vertex-program Q(v), which can be executed in parallel on the vertices and can interact with adjacent vertices.
- **G**ather – information from adjacent vertices/edges is reduced by a generalized sum (commutative and associative).

$$\Sigma \leftarrow \bigoplus_{v \in \mathbf{Nbr}[u]} g\left(D_u, D_{(u,v)}, D_v\right)$$

GAS Model

- **A**pply – the gathered sum is used with the current value to update the current vertex value

$$D_u^{\text{new}} \leftarrow a\left(D_u, \Sigma\right)$$

- Scatter – the new value is used to update data on adjacent edges

$$\forall v \in \mathbf{Nbr}[u] : \quad \left(D_{(u,v)}\right) \leftarrow s\left(D_u^{\text{new}}, D_{(u,v)}, D_v\right)$$

Pregel

- Bulk-Synchronous execution – all vertex-programs run simultaneously in lock step – within each “super-step,” each Q(v) receives messages from the previous step and sends to the next step. It terminates when there are no remaining messages and all programs vote to stop.

- Combiners – Associative and Commutative functions that merge incoming messages from neighboring vertices (no edge data).

```
Message combiner(Message m1, Message m2) :
  return Message(m1.value() + m2.value());
void PregelPageRank(Message msg) :
  float total = msg.value();
  vertex.val = 0.15 + 0.85*total;
  foreach(nbr in out_neighbors) :
     SendMsg(nbr, vertex.val/num_out_nbrs);
```

- Gather = combiners, Apply/Scatter = vertex-program

GraphLab

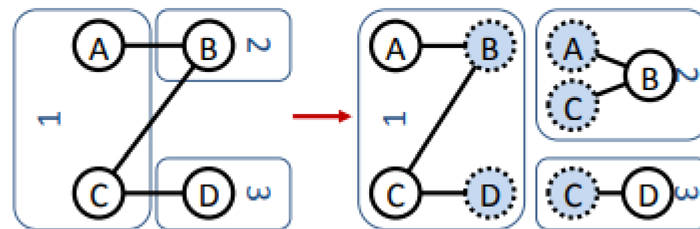
- Asynchronous Distributed Shared-Memory Execution – all vertex programs access a distributed graph with the current data, each can access the adjacent vertices and edges. Vertex-programs schedule their neighbors to run in the future (serializability by preventing neighboring instances from running simultaneously).
- There is no messaging – vertices directly access the data they need.

```
void GraphLabPageRank(Scope scope) :
  float accum = 0;
  foreach (nbr in scope.in_nbrs) :
     accum += nbr.val / nbr.nout_nbrs();
  vertex.val = 0.15 + 0.85 * accum;
```

- Gather/Apply = vertex-program, Scatter = direct access (changes are immediately visible to neighbors)
- GraphLab provides strong serializability (every parallel execution corresponds to a sequential execution) but does this through a sequential locking protocal that is unfair to high-degree vertices

Edge-Cuts

- The existing method to partition the graph – vertices are evenly assigned to machines such that the number of "cut edges" (edges spanning machines) is minimized.
- If adjacent vertices A, B are on distinct machines 1, 2, then they use "ghost" vertices (e.g. machine 1 has a ghost vertex B, and machine 2 has a ghost vertex A) – changes have to be synchronized to ghosts.
- Balanced edge-cut algorithms perform poorly on power-law graphs, so GraphLab and Pregel both use randomized placement – this is bad.



(a) Edge-Cut

Randomized Edge-Cuts

Theorem 5.1. *If vertices are randomly assigned to p machines then the expected fraction of edges cut is:*

$$\mathbb{E}\left[\frac{|\textit{Edges Cut}|}{|E|}\right] = 1 - \frac{1}{p}. \tag{5.1}$$

For a power-law graph with exponent α, the expected number of edges cut per-vertex is:

$$\mathbb{E}\left[\frac{|\textit{Edges Cut}|}{|V|}\right] = \left(1 - \frac{1}{p}\right)\mathbb{E}\left[\mathbf{D}[v]\right] = \left(1 - \frac{1}{p}\right)\frac{\mathbf{h}_{|V|}(\alpha - 1)}{\mathbf{h}_{|V|}(\alpha)}, \tag{5.2}$$

where the $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law Zipf distribution.

Proof. An edge is cut if both vertices are randomly assigned to different machines. The probability that both vertices are assigned to different machines is $1 - 1/p$. □

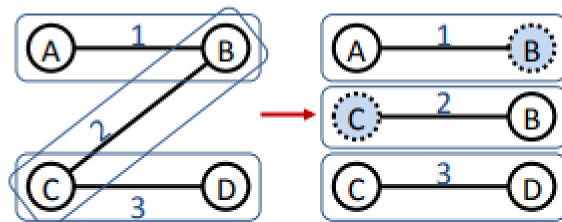
- So, the communication costs are nearly linear in |E| (to synchronize across ghosts).

Challenges of Natural Graphs

- The key issue is that a small number of vertices have a large number of the edges (e.g. 1% of vertices in Twitter graph have ~50% of edges)
- Work Balance: Existing graph-parallel frameworks treat vertices symmetrically and have storage/communication/computation costs linear in degree
- Partitioning: GraphLab and Pregel both depend on partitioning the graph – this is hard to do in natural graphs, so they both end up using random partitioning, which is bad (seen later)
- Communication/Storage: Because of the skewed distribution, there are major bottlenecks in communication and heavy memory usage at high-degree vertices
- Computation: Existing frameworks do not parallelize individual vertex-programs, limiting their scalability in skewed graphs

Vertex-Cuts

- The intuition here is that we can either cut the vertices or the edges to partition the graph. The distribution of vertex degree is highly skewed, but the distribution of number of vertices adjacent to a given edge is constant (e.g. it's always 2). Thus, cutting vertices has much better potential.
- In the vertex-cut, every edge is assigned to an individual machine, and the vertices are cut across machines.
- Each vertex is replicated across the machines where its adjacent edges lie – the communication overhead is the total sum of the number of vertex replicas. Replicas are referred to as mirrors.



(b) Vertex-Cut

Vertex-Cuts

- The formal objective is to minimize communication costs by minimizing the total vertex replication.

A balanced p-way **vertex-cut** formalizes this objective by assigning each *edge* $e \in E$ to a machine $A(e) \in \{1,\ldots,p\}$. Each vertex then spans the set of machines $A(v) \subseteq \{1,\ldots,p\}$ that contain its adjacent edges. We define the balanced vertex-cut objective:

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)| \tag{5.3}$$

$$\text{s.t.} \quad \max_{m} |\{e \in E \mid A(e) = m\}|, < \lambda \frac{|E|}{p} \tag{5.4}$$

where the imbalance factor $\lambda \geq 1$ is a small constant. We

Randomized Vertex-Cuts

Theorem 5.2 (Randomized Vertex Cuts). *A random vertex-cut on p machines has an expected replication:*

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v\in V}|A(v)|\right] = \frac{p}{|V|}\sum_{v\in V}\left(1-\left(1-\frac{1}{p}\right)^{\mathbf{D}[v]}\right). \quad (5.5)$$

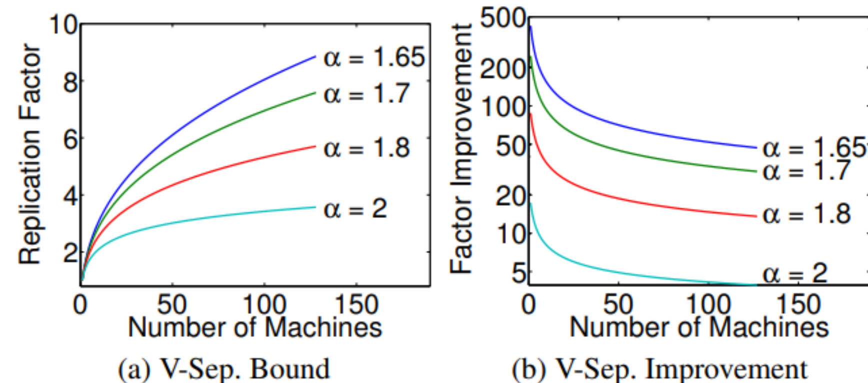
where $\mathbf{D}[v]$ *denotes the degree of vertex v. For a power-law graph the expected replication (Fig. 6a) is determined entirely by the power-law constant* α*:*

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v\in V}|A(v)|\right] = p - \frac{p}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}\left(\frac{p-1}{p}\right)^{d} d^{-\alpha}, \quad (5.6)$$

where $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ *is the normalizing constant of the power-law Zipf distribution.*

Randomized Vertex-Cuts

- The randomized vertex-cut achieves very good expected performance, getting almost perfect balance across the machines.
- Lower α causes higher replication factors, but also gives higher effective gains compared to a random edge-cut (in practice).



(a) V-Sep. Bound (b) V-Sep. Improvement

Figure 6: **(a)** Expected replication factor for different power-law constants. **(b)** The ratio of the expected communication and storage cost of random edge cuts to random vertex cuts as a function of the number machines. This graph assumes that edge data and vertex data are the same size.

Greedy Vertex-Cuts

$$\arg\min_{k} \mathbb{E}\left[\sum_{v\in V} |A(v)| \,\middle|\, A_i, A(e_{i+1}) = k\right]$$

Case 1: If $A(u)$ and $A(v)$ intersect, then the edge should be assigned to a machine in the intersection.

Case 2: If $A(u)$ and $A(v)$ are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.

Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.

Case 4: If neither vertex has been assigned, then assign the edge to the least loaded machine.

Greedy Vertex-Cuts

- The optimally greedy solution has a high overhead to communicate and coordinate between machines, so there are two different implementations: a **coordinated** implementation that uses a shared table to maintain A values, and an **oblivious** implementation that runs independently on each machine and estimates neighboring A values.
- In general, the oblivious implementation provides a good tradeoff between the cost of calculating the vertex-cut and the replication benefits provided by the resulting partition.

Vertex-Cut Comparisons

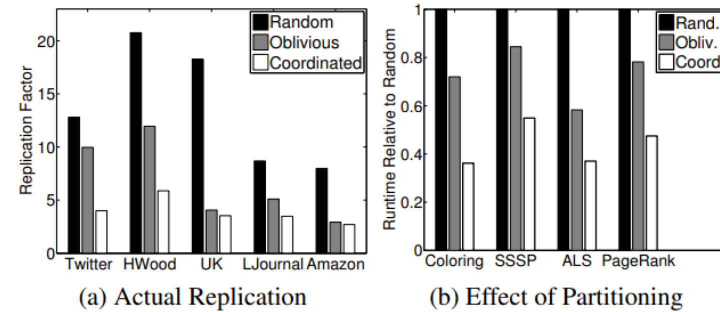


Figure 7: **(a)** The actual replication factor on 32 machines. **(b)** The effect of partitioning on runtime.

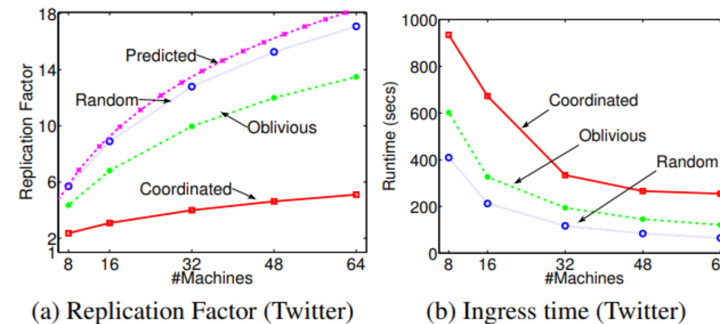


Figure 8: **(a,b)** Replication factor and runtime of graph ingress for the Twitter follower network as a function of the number of machines for random, oblivious, and coordinated vertex-cuts.

PowerGraph: General Idea

- Combines aspects of Pregel and GraphLab but uses vertex-cuts to partition the graph.
- From Pregel, it takes the commutative/associative gather, which allows for gathering to be done by the framework rather than the vertex-program. It also takes the bulk-synchronous execution model.
- From GraphLab it takes the shared-memory data-graph, so that the framework can handle communication. It also takes the asynchronous computation model.
- The framework introduces a stateless vertex-program through a GASVertexProgram interface. The gather, sum, apply, and scatter are explicitly separated, and the communication is all handled by the framework, rather than the vertex-program.

PowerGraph: Abstraction

- Gather, Sum act as a map-reduce processor on the adjacent data.
- Apply is done on the sum and current vertex's value and atomically writes back to the data-graph. In order for the framework to be effective, the work should be sublinear (ideally constant) in the degree of the node.
- Scatter is invoked in parallel and produces both an edge value and delta value (explained later).

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_(u,v), D_v) → (D_(u,v)^new, Accum)
}
```

Figure 2: All PowerGraph programs must implement the stateless gather, sum, apply, and scatter functions.

Algorithm 1: Vertex-Program Execution Semantics

Input: Center vertex u
if *cached accumulator* a_u *is empty* **then**
  foreach *neighbor* v *in gather_nbrs(u)* **do**
    $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{(u,v)}, D_v))$
  end
end
$D_u \leftarrow \text{apply}(D_u, a_u)$
foreach *neighbor* v *scatter_nbrs(u)* **do**
  $(D_{(u,v)}, \Delta a) \leftarrow \text{scatter}(D_u, D_{(u,v)}, D_v)$
  if a_v *and* Δa *are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$
  else $a_v \leftarrow \text{Empty}$
end

PowerGraph: Examples

PageRank

```
// gather_nbrs: IN_NBRS
gather(D_u, D_(u,v), D_v):
  return D_v.rank / #outNbrs(v)
sum(a, b): return a + b
apply(D_u, acc):
  rnew = 0.15 + 0.85 * acc
  D_u.delta = (rnew - D_u.rank)/
           #outNbrs(u)
  D_u.rank = rnew
// scatter_nbrs: OUT_NBRS
scatter(D_u,D_(u,v),D_v):
  if(|D_u.delta|>ε) Activate(v)
  return delta
```

Greedy Graph Coloring

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return set(D_v)
sum(a, b): return union(a, b)
apply(D_u, S):
  D_u = min c where c ∉ S
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // Nbr changed since gather
  if(D_u == D_v)
    Activate(v)
  // Invalidate cached accum
  return NULL
```

Single Source Shortest Path (SSSP)

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return D_v + D_(v,u)
sum(a, b): return min(a, b)
apply(D_u, new_dist):
  D_u = new_dist
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // If changed activate neighbor
  if(changed(D_u)) Activate(v)
  if(increased(D_u))
    return NULL
  else return D_u + D_(u,v)
```

Figure 3: The PageRank, graph-coloring, and single source shortest path algorithms implemented in the PowerGraph abstraction. Both the PageRank and single source shortest path algorithms support delta caching in the gather phase.

PowerGraph: Vertex-Cuts

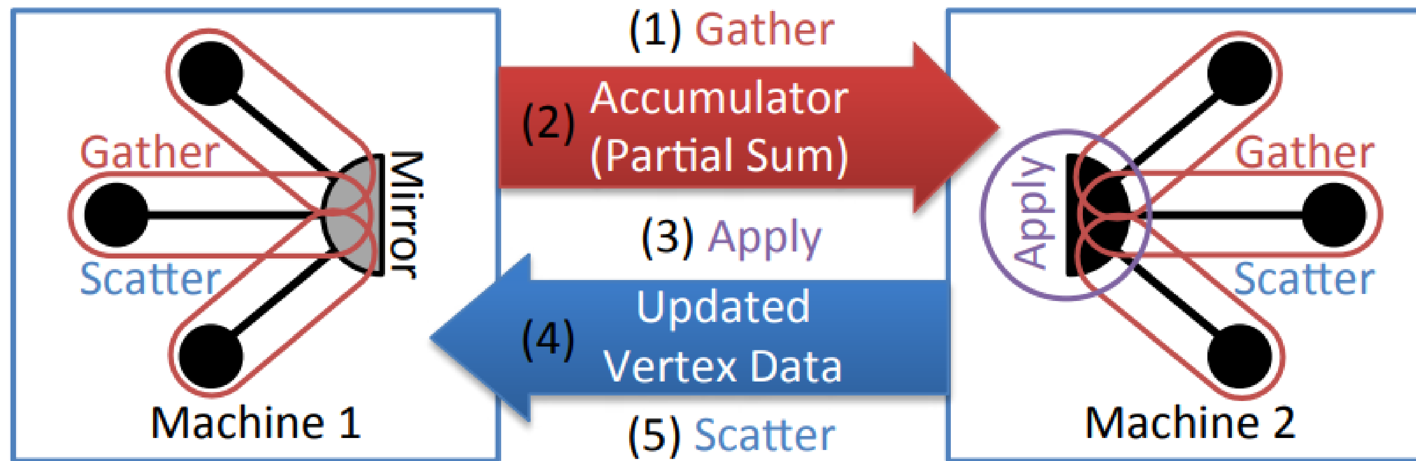


Figure 5: The communication pattern of the PowerGraph abstraction when using a vertex-cut. Gather function runs locally on each machine and then one accumulators is sent from each mirror to the master. The master runs the apply function and then sends the updated vertex data to all mirrors. Finally the scatter phase is run in parallel on mirrors.

PowerGraph: Delta Caching

- For further optimization, the framework provides delta caching
- Many times, only a few of the vertex-program's neighbors have changed from the previous run. Thus, the accumulator values are cached at each vertex, and the scatter function can return a delta value to directly apply to the neighboring cached accumulator. If this value is not returned, the neighboring cache is cleared.

Intuitively, Δa acts as an additive correction on-top of the previous gather for that edge. More formally, if the accumulator type forms an **abelian group**: has a commutative and associative sum $(+)$ and an *inverse* $(-)$ operation, then we can define (shortening `gather` to g):

$$\Delta a = g(D_u, D_{(u,v)}^{\text{new}}, D_v^{\text{new}}) - g(D_u, D_{(u,v)}, D_v). \qquad (4.1)$$

PowerGraph: Execution Model

- PowerGraph provides three modes of computation: bulk-synchronous, asynchronous, and serializable asynchronous.
- The bulk-synchronous model splits the execution into gather, apply, and scatter phases (“minor-steps”). Each of these minor-steps are run synchronously on all active vertices in lock-step, and changes are committed at the end of each minor-step (and visible on the next). Activated vertices are added to the next super-step (a full GAS series).
- In asynchronous execution, changes are immediately visible to other vertices and vertices are run as soon as resources are available. This is inherently non-deterministic, so PowerGraph provides a strong serializability guarantee by using parallel locking that is fair to high-degree vertices.

Comparisons

- Both Pregel and GraphLab can be simulated in PowerGraph by writing the appropriate vertex-programs. However, this obviates the benefits of PowerGraph.
- PowerGraph's great strength is when the apply function is sub-linear in the degree of the vertex. Then, the gather and scatter steps are executed in parallel across all the edges (on different machines), and the actual vertex computation is done very quickly on a single master vertex.
- PowerGraph has very well-balanced computation and communication costs as well as runtime because of the vertex-cut properties.

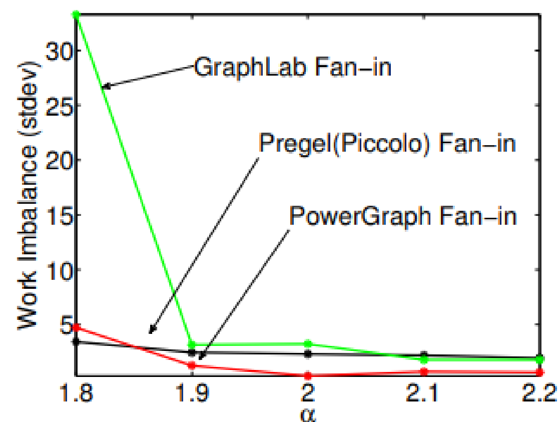
Comparisons

Theorem 5.3. *For a given an edge-cut with g ghosts,* ***any*** *vertex cut along the same partition boundary has strictly fewer than g mirrors.*

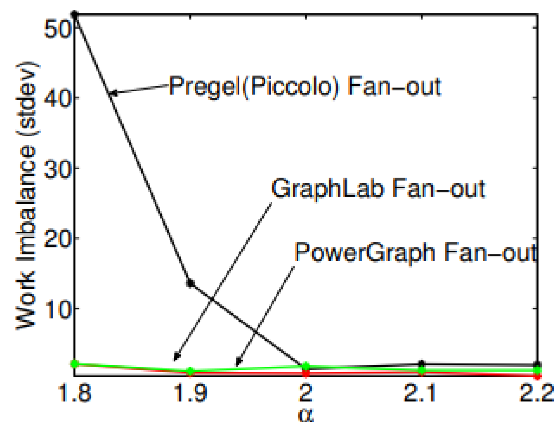
Proof of Theorem 5.3. Consider the two-way edge cut which cuts the set of edges $E' \in E$ and let V' be the set of vertices in E'. The total number of ghosts induced by this edge partition is therefore $|V'|$. If we then select and delete arbitrary vertices from V' along with their adjacent edges until no edges remain, then the set of deleted vertices corresponds to a vertex-cut in the original graph. Since at most $|V'| - 1$ vertices may be deleted, there can be at most $|V'| - 1$ mirrors. □

Comparisons

- The three frameworks were compared by running PageRank over synthetic power-law graphs. For each value of α, the graphs were generated using the power-law for out-degree (fan-out) and maintaining roughly equal in-degrees and then by inverting these to get power-law in-degree (fan-in) graphs.
- For computation, the Pregel implementation is linear in out-degree and the GraphLab implementation is linear in in-degree. On the other hand, PowerGraph has no degree dependence in runtime because the gather/scatter are spread across machines.



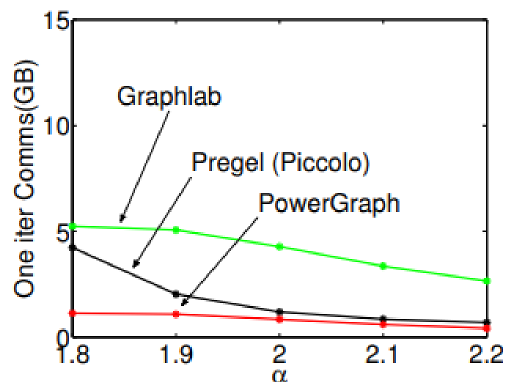
(a) Power-law Fan-In Balance



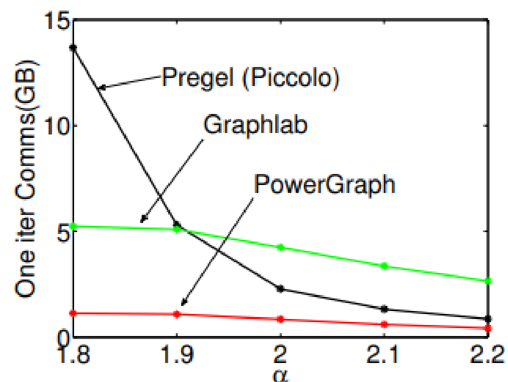
(b) Power-law Fan-Out Balance

Comparisons

- For communication, both GraphLab (fan-in gather) and Pregel (fan-out scatter) have costs proportional to the number of ghosts – the data must be sent across any edges that span machines. On the other hand, PowerGraph communication is proportional to the number of mirrors, which we showed is smaller in expectation.
- Pregel sends messages on fan-out edges, but GraphLab is symmetric to fan-in and fan-out edges.



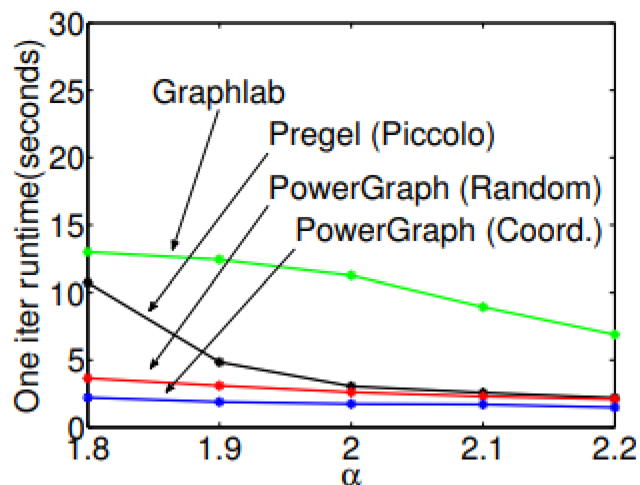
(c) Power-law Fan-In Comm.



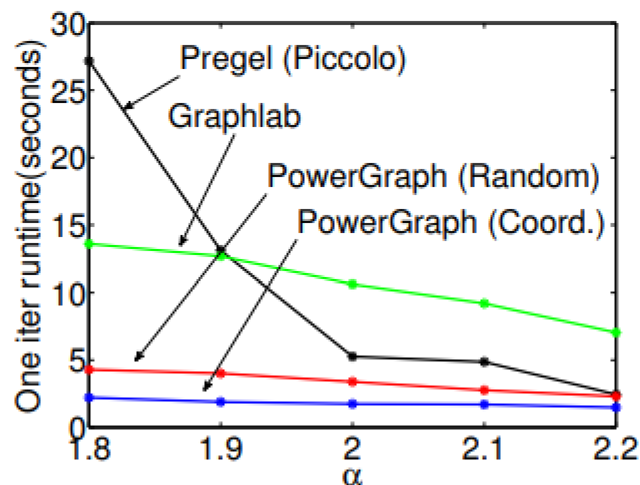
(d) Power-law Fan-Out Comm.

Comparisons

- In overall runtime, the pattern basically follows the shapes we saw in the communication imbalance graphs, because PageRank is computationally simple so the runtime is dominated by messaging.
- Greedy partitioning yields even better performance for PowerGraph.



(a) Power-law Fan-In Runtime



(b) Power-law Fan-Out Runtime

Results

- Run with HDFS providing the backing data store. Each node reads a unique set of data files and loaded into position in parallel with the partitioning algorithm
- The general experiment is PageRank on Twitter with oblivious partitioning (unless otherwise specified).
- Runtime scaled linearly with replication.

Synchronous Results

- For any reasonably large job, coordinated partitioning gave great runtime benefits
- Delta caching reduced runtime by 45%
- Nearly optimal (linear) weak scaling – job size per processor remains constant as processors are added.

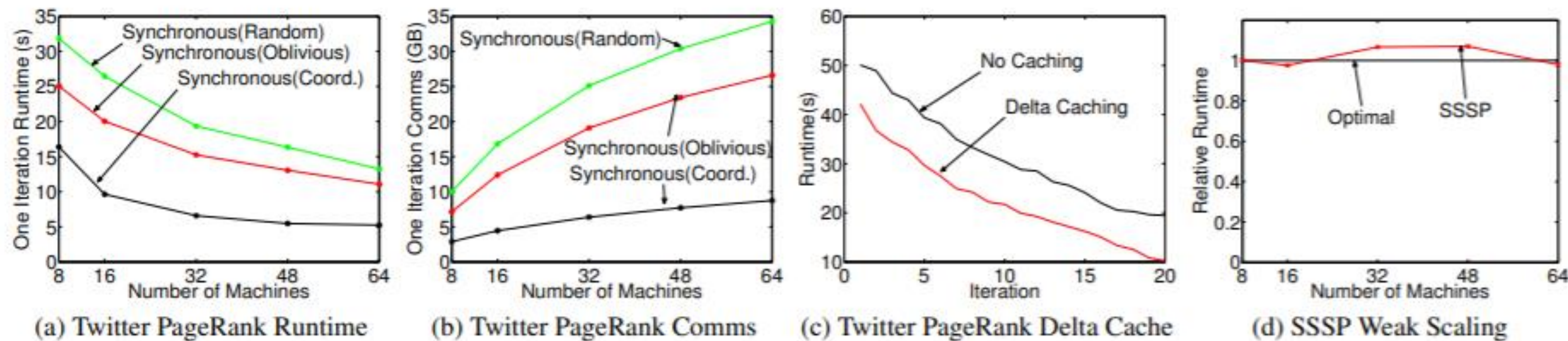
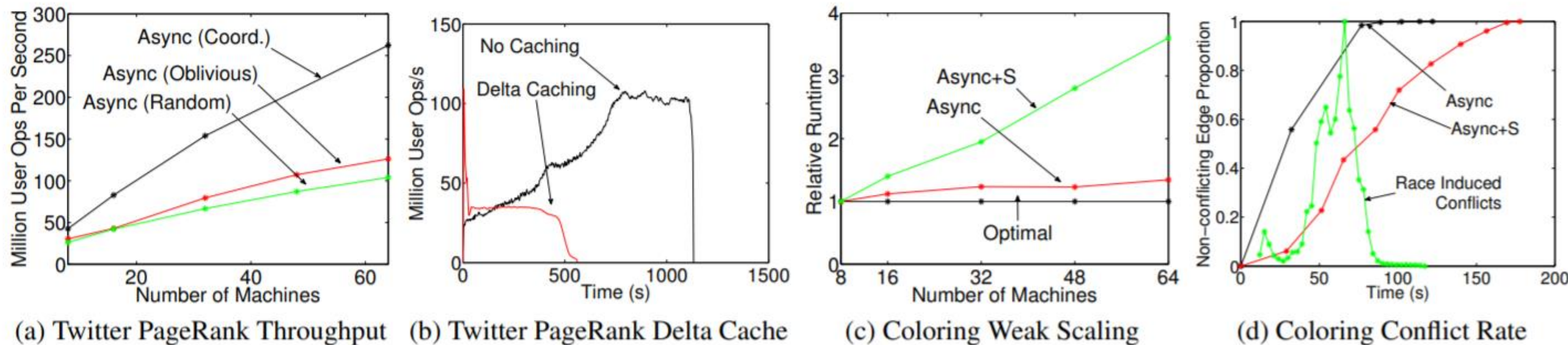


Figure 11: **Synchronous Experiments (a,b)** Synchronous PageRank Scaling on Twitter graph. **(c)** The PageRank per iteration runtime on the Twitter graph with and without delta caching. **(d)** Weak scaling of SSSP on synthetic graphs.

Asynchronous Results

- In regular asynchronous mode (Async), avoid data races by ensuring exclusive access to arguments.
- When in serializable mode (Async + S), ensure serializability by preventing adjacent programs from running simultaneously.
  - They use the Chandy-Misra solution to dining philosophers to accomplish this because it is more parallel than other solutions (acquire locks simultaneously)
- In Async, with caching disabled, throughput increases with time because the computation becomes focused on high-degree nodes.
- Async has essentially linear weak scaling while Async+S does not (because the contention grows super-linearly with problem size).
- Serializability shows its value in the graph-coloring problem, where Async spends 34% of the runtime coloring the final 1% of edges (contested) while Async colors the full graph with half the work.

Asynchronous Results

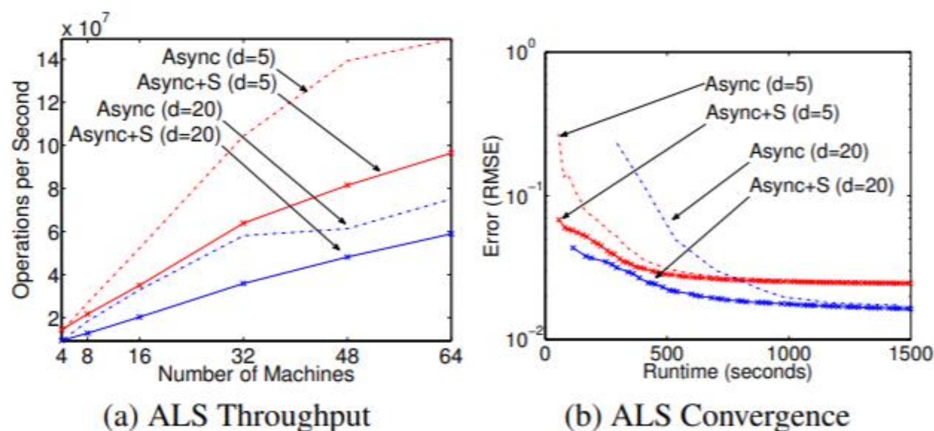


(a) Twitter PageRank Throughput (b) Twitter PageRank Delta Cache (c) Coloring Weak Scaling (d) Coloring Conflict Rate

Figure 12: **Asynchronous Experiments (a)** Number of user operations (gather/apply/scatter) issued per second by Dynamic PageRank as # machines is increased. **(b)** Total number of user ops with and without caching plotted against time. **(c)** Weak scaling of the graph coloring task using the Async engine and the Async+S engine **(d)** Proportion of non-conflicting edges across time on a 8 machine, 40M vertex instance of the problem. The green line is the rate of conflicting edges introduced by the lack of consistency (peak 236K edges per second) in the Async engine. When the Async+S engine is used no conflicting edges are ever introduced.

Asynchronous Results

- Alternating Least Squares (ALS) workload – ML algorithm, complexity determined by parameter d ($O(d^3)$).
- Note that Async+S has lower throughput but converges faster – serializability is often required for convergence
- The difference in throughput decreases as d increases (runtime shifts from communication to computation).



(a) ALS Throughput (b) ALS Convergence

Figure 13: **(a)** The throughput of ALS measured in millions of User Operations per second. **(b)** Training error (lower is better) as a function of running time for ALS application.

Results

- Fault tolerance through snapshotting.
- Some other comparisons against real systems. Note that Smola et al.'s LDA solution is very complicated, special-purpose code to solve that exact problem, and it achieves similar performance to PowerGraph with ~200 lines of user code.

PageRank	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [22]	198s	–	1.1B	50x8
Spark [37]	97.4s	40M	1.5B	50x2
Twister [15]	36s	50M	1.4B	64x4
PowerGraph (Sync)	3.6s	40M	1.5B	64x8

Triangle Count	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [36]	423m	40M	1.4B	1636x?
PowerGraph (Sync)	1.5m	40M	1.4B	64x16

LDA	Tok/sec	Topics	System
Smola et al. [34]	150M	1000	100x8
PowerGraph (Async)	110M	1000	64x16

Table 2: Relative performance of PageRank, triangle counting, and LDA on similar graphs. PageRank runtime is measured per iteration. Both PageRank and triangle counting were run on the Twitter follower network and LDA was run on Wikipedia. The systems are reported as number of nodes by number of cores.

Related Work

- There are many existing graph-parallel frameworks, and their main properties are represented throughout the presentation by Pregel and GraphLab.
- Some others are BPGL (similar to Pregel), Kineograph (somewhat similar to GraphLab and Pregel)
- Vertex-cut is similar to hypergraph partitioning (edge->vertex, vertex->hyper edge).
- The streaming (greedy) vertex cut is a novel innovation, but there are existing streaming edge-cut algorithms (Stanton et al).
- GraphChi is an efficient single-machine implementation of the GraphLab abstraction that could be used in conjunction with PowerGraph to provide “out-of-core storage.”

Thoughts

- Strengths
  - This paper has many novel ideas, especially the vertex-cut and its efficient computation, and leverages them to build a power framework
  - It also nearly abstracts the general problem and the framework to demonstrate its applicability to various classes of problems
  - It manages to parallelize operations with relatively low synchronization overhead, allowing for high scalability.
  - It outperforms state-of-the-art graph-parallel frameworks
- Weaknesses
  - This paper presented the ideas in an odd order that made it harder to understand in a single pass. In particular, the vertex-cut idea should be presented before the PowerGraph abstraction.
  - The paper sometimes focuses more on comparisons than on the idea itself.

Discussion

- What are the key ideas that are presented in this paper and how could they be applicable beyond graph-parallel frameworks?
- Why do you think the other frameworks were developed without explicit attention towards natural graphs?
- For vertex cuts, does the edge imbalance across machines matter more or less than the replication factor?
- Would it be beneficial to consider the balance of edges for each vertex across machines (i.e. to ensure equitable distribution of work for every individual vertex)?