Systems and Methods for Exascale Graph Analytics

presented by **Mohammad Mofrad**
University of Pittsburgh

April 27, 2018

Comprehensive exam committee

Professor Rami Melhem, Computer Science Department, University of Pittsburgh

Professor Alex Labrinidis, Computer Science Department, University of Pittsburgh

Professor Jack Lange, Computer Science Department, University of Pittsburgh

Discussion Outline

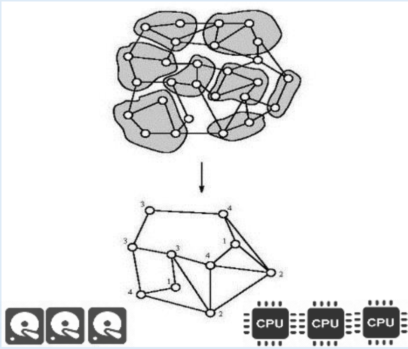
Graph Partitioning

Vertex-centric, architecture-aware and streaming

Cloud-based Graph Analytics Platforms

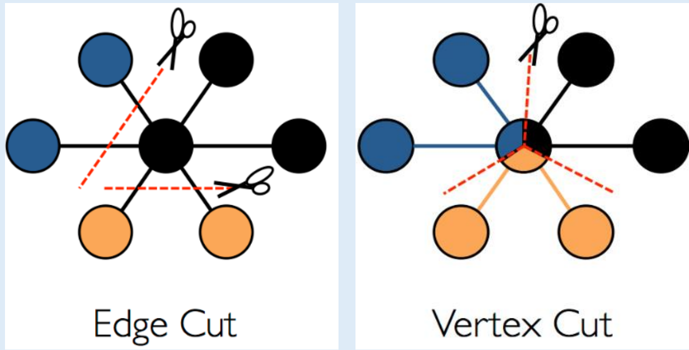
HPC-based Graph Analytics Platforms

Graph Partitioning Goals and Metrics



- **Partitioning**
  - Random or Hash-based partitioning have extremely poor *locality* and *cut-edge*
- **Work balance**
  - When: **partition → Node in the cluster**
  - Scalability limitation for high degree vertices
  - *Symmetric computation* at vertices
- **Computation**
  - Exploiting *higher parallelism*
  - Distributing computation
  - Edges or vertices
- **Communication**
  - Communication asymmetry
- **Storage**
  - Aggregating *storage* mediums across machines
  - Exceeding memory capacity

- ***k*-way balanced partitioning** of **G** = (V, E)
  - $|E| / k \,.\, (1 + \varepsilon)$ i.e. $\varepsilon > 0$
  - $|V| / k \,.\, (1 + \varepsilon)$ i.e. $\varepsilon > 0$
- Partitioning criteria:
  - *Edge cut*
  - *Vertex cut*



J. E. Gonzalez, et al. "PowerGraph: Distributed graph-parallel computation on natural graphs." OSDI, 2012.

Discussion Outline

Graph Partitioning

- *Architecture-aware* (**Aragon, Paragon, Planar and Argo**)
- Vertex-centric
- Streaming

Cloud-based Graph Analytics Platforms

HPC-based Graph Analytics Platforms

Architecture Aware Graph Partitioning

- *Non-uniform Inter-node communication*
  - Communication cost among nodes
- *Non-uniform Intra-node communication*
  - Cache hierarchy among cores

- *Migration cost*
  - Among nodes
    - Because of network interconnect
  - Among cores
    - Because of memory hierarchy

GOAL: (Re)balance the load across nodes while minimizing inter-node communication and migration cost (not just edge-cut)

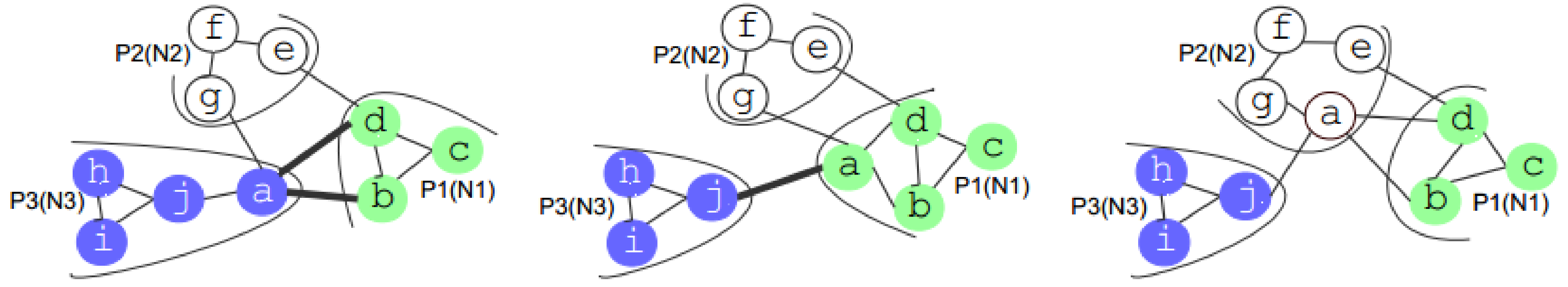


Figure 3: Old Decomposition

Figure 4: Better Decomposition

Figure 5: Best Decomposition

	N_1	N_2	N_3
N_1		1	6
N_2	1		1
N_3	6	1	

Figure 6: Relative Network Communication Costs

Lower cut-edge

Better communication

A. Zheng, et al. "Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing." Big Data, 2014.

Architecture Aware Graph Partitioning

- Problem Statement: Let **G** = (*V*, *E*)

$$P = \{P_i : \cup_i^n P_i = V \text{ and } P_i \cap P_j = \phi \text{ for any } i \neq j\}$$

An unbalanced partitioning of G

- Balance the *load*

$$w(P_i) < (1+\varepsilon) * \bar{w} \qquad \bar{w} = \frac{\sum_{j=1}^{n} w(P_j)}{n}$$

$w(P_i)$ is the aggregated weight of vertices
ε is the imbalanced ratio

- Minimize the *communication cost*

$$comm(G, P') = \alpha * \sum_{\substack{e=(u,v)\in E \\ \text{and } u\in P'_i \text{ and } v\in P'_j \text{ and } i\neq j}} w(e) * c(P'_i, P'_j)$$

α is the #steps
$w(e)$ is the edge weight
$c(P'_i, P'_j)$ is the communication cost

- Minimize the *migration cost*

$$mig(G, P, P') = \sum_{v\in P_i \text{ and } v\in P'_j \text{ and } i\neq j} vs(v) * c(P_i, P'_j)$$

$vs(v)$ is the vertex size
$c(P_i, P'_j)$ is the migration cost

A. Zheng, et al. "Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing." Big Data, 2014.

Aragon: Two Phase Partitioning

- **1. Cluster**, 2. Cores
  - Inter-node partitioning (Comparison)
    - **TopoFM**: (2 partitions + communication cost) → Repartition
      - Process a single vertex per iteration!
- Topology aware Gain computation $g(v)$
  - P_i and P_j partitions are placed in N_i and N_j nodes with $v \in P_i$
    - Greedy gain function

$$g_{std}(v) = \alpha * (d^j_{ext}(v) - d^i_{int}(v)) * d(N_i, N_j)$$

$$d^i_{int}(v) = \sum_{e=(v,u)\in E \text{ and } v\in P_i \text{ and } u\in P_i} w(e)$$

$$d^j_{ext}(v) = \sum_{e=(v,u)\in E \text{ and } v\in P_i \text{ and } u\in P_j \text{ and } i\neq j} w(e)$$

$$g_{topo}(v) = \alpha * \sum_{e=(v,u)\in E \text{ and } v\in P_i \text{ and } u\in P_k \text{ and } k\neq i \text{ and } k\neq j} w(e)*(d(N_i, N_k)-d(N_j, N_k))$$

$$g_{mig}(v) = vs(v) * (d(N_i, N_k) - d(N_j, N_k))$$

$$g(v) = g_{std}(v) + g_{topo}(v) + g_{mig}(v)$$

A. Zheng, et al. "Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing." Big Data, 2014.

Aragon: Two Phase Partitioning

- 1. Cluster, **2. Cores**
  - Intra-node partitioning
    - **HierCacheLB** (*Partition hierarchically*)
    - **FlatCacheLB** (partition entirely and then assign)
  - Advantages
    - Consider both network topology and system architecture at the same time
    - Most works that I read consider communication is cheap
  - **Drawbacks**
    - Memory hug
    - Uniform hardware layout
    - Can only refine one partition at a time, so it is a sequential algorithm

Tree Communication cost

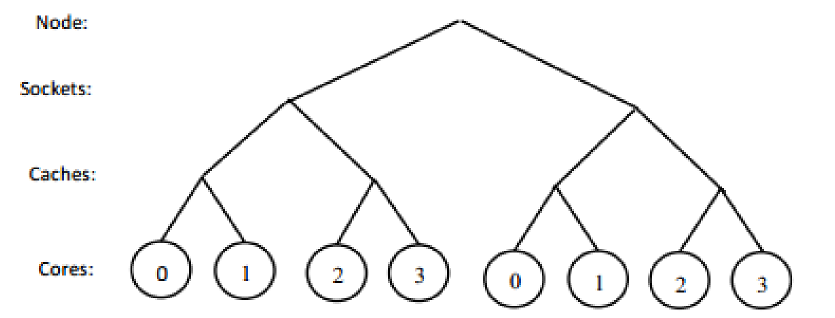
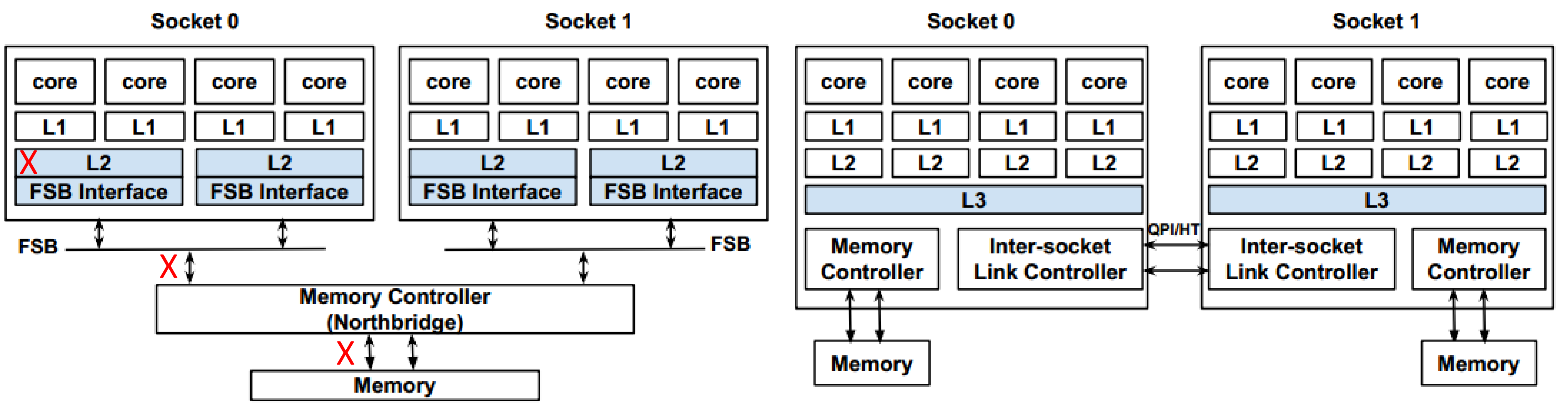


Fig. 5: Topology Tree

A. Zheng, et al. "Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing." Big Data, 2014.

Argo: The Curse of Contention



(a) Uniform Memory Access (UMA) Node

(b) Nonuniform Memory Access (NUMA) Node

Data communication among cores is done via shared memory which is a source of contention

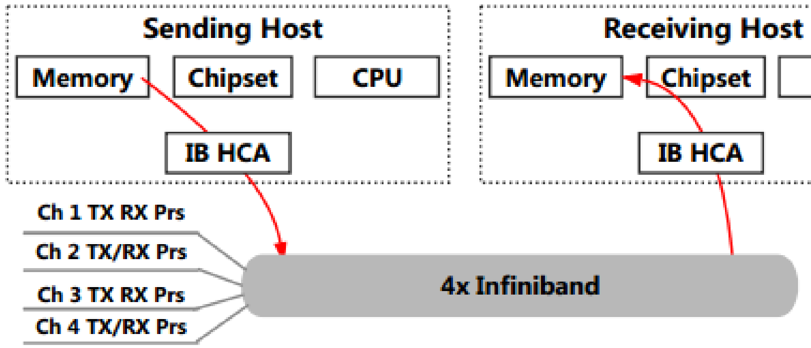


Fig. 2: Memory transactions of inter-node data communication via RDMA [14]

Zero-copy without involvement

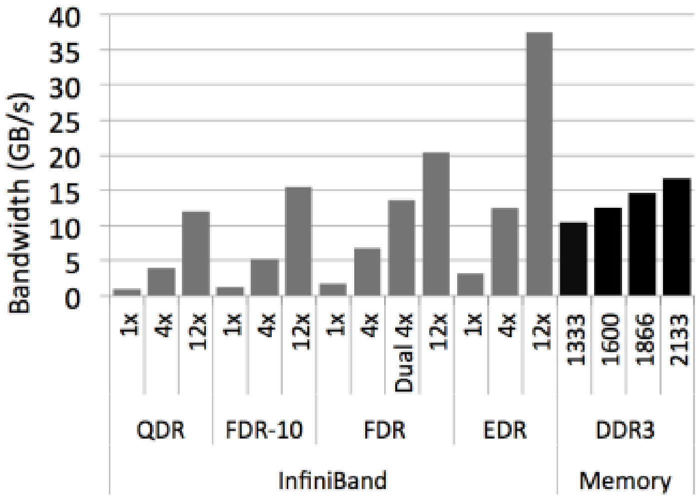


Fig. 3: Theoretic bandwidth for different InfiniBand and memory technologies (Binnig et. al. [9].)

A. Zheng, et al. "Argo: Architecture-aware graph partitioning." Big Data, 2016.

Argo: Graph Partitioning Model

- Derived from *linear deterministic greedy algorithm*
  - A streaming partitioning algorithm
- Argo (with *heterogeneity awareness*)

$$\boxed{(1 - \frac{w(P_i)}{C(P_i)})} * \boxed{\sum_{e=(u,v)\in E \text{ and } u\in P_i} w(e)}$$

$$\frac{1}{comm(v,P_i)+1} * (1 - \frac{w(P_i)}{C(P_i)})$$

$$comm(v,P_i) = \sum_{e=(u,v)\in E \text{ and } u\in P_j \text{ and } i\neq j} w(e) * c(P_i,P_j)$$

$$c(P_i,P_j) = c(P_i,P_j) + \lambda * (s_1 + s_2)$$

- Contention awareness
  - Penalize intra-node communication by offloading a certain amount of intra-node communication across compute nodes
  - s_1 and s_2 are inter-node and inter-socket communication costs
  - $\lambda \in [0, 1]$ controls the communication & contention heterogeneity
    - $\lambda = 0$ only communication; $\lambda = 1$ only contention; $\in (0, 1]$ both

A. Zheng, et al. "Argo: Architecture-aware graph partitioning." Big Data, 2016.

Aragon, Paragon, Planar, and Argo Comparison

Features	Aragon	Paragon	Planar	Argo
Architecture-aware	Yes	Yes	Yes	Yes
Algorithm	Sequential	Parallel	Parallel & Adaptive	Parallel
Runtime	Heavyweight	Lightweight	Lightweight	Lightweight
Incremental	No	No	Yes	No
Partitioning Space	All	Boundary vertices	Boundary vertices	All
Balanced partitions	Edge weights	Edge weights	Edge weights	Edge weights
Migration decision	Deterministic	Deterministic	Probabilistic ($p \in [0, max_g]$)	Greedy
Speed	Slow	Decent	Fast	Fast
Resource Contention	No	Yes	Yes	Yes

A. Zheng, et al. "Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing." Big Data, 2014.
A. Zheng, et al. "Paragon: Parallel Architecture-Aware Graph Partition Refinement Algorithm ." EDBT, 2016.
A. Zheng, et al. "Planar: Parallel Lightweight Architecture-Aware Adaptive Graph Repartitioning." ICDE, 2016.
B. A. Zheng, et al. "Argo: Architecture-aware graph partitioning." Big Data, 2016.

Discussion Outline

Graph Partitioning

- Architecture-aware
- *Vertex-centric* (**Spinner** and **Ja-Be-Ja**)
- *Streaming* (**Fennel**)

Cloud-based Graph Analytics Platforms
HPC-based Graph Analytics Platforms

Spinner: Balanced k-way label propagation

$$score(v,l) = \sum_{u \in N(v)} \delta(\alpha(u), l)$$

$$l_v = \arg\max_l \, score(v,l)$$

$$score'(v,l) = \sum_{u \in N(v)} w(u,v)\delta(\alpha(u),l)$$

$$w(u,w) = \begin{cases} 1, & \text{if } (u,v) \in D \oplus (v,u) \in D \\ 2, & \text{if } (u,v) \in D \wedge (v,u) \in D \end{cases}$$

$$score''(v,l) = \sum_{u \in N(v)} \frac{w(u,v)\delta(\alpha(u),l)}{\sum_{u \in N(v)} w(u,v)} - \pi(l)$$

$$\pi(l) = \frac{b(l)}{C} \qquad b(l) = \sum_{v \in G} deg(v)\delta(\alpha(v),l) \qquad C = c \cdot \frac{|E|}{k}$$

- Migration decisions

$$p = \frac{r(l)}{m(l)} \qquad r(l) = C - b(l) \qquad m(l) = \sum_{v \in M(l)} deg(v)$$

- Evaluation metrics

$$\phi = \frac{\#\ local\ edges}{|E|}, \ \rho = \frac{maximum\ load}{\frac{|E|}{k}}$$

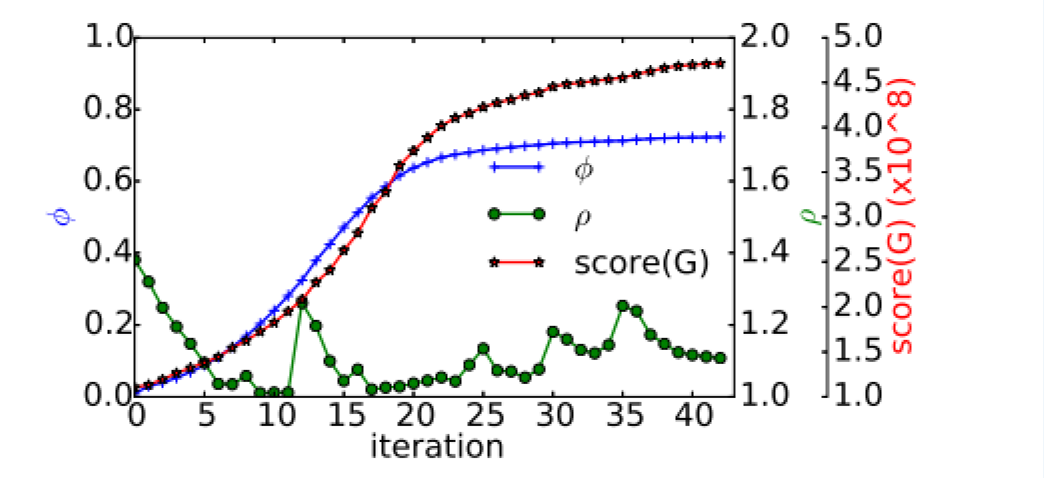
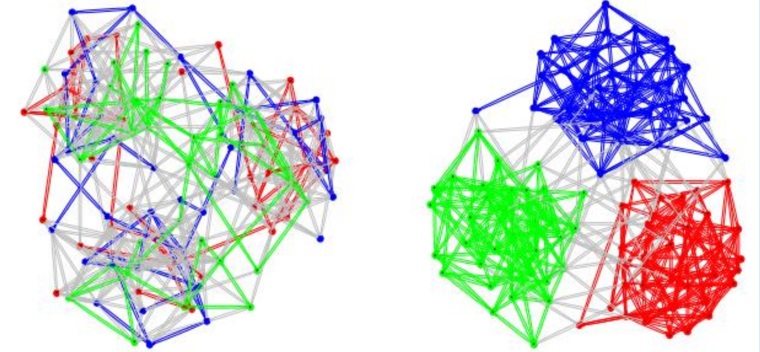


Fig. 3. Partitioning of the Yahoo! web graph across 115 partitions. The figure shows the evolution of metrics ϕ, ρ, and $score(G)$ across iterations.

C. Martella, et al. "Spinner: Scalable graph partitioning in the cloud." ICDE, 2017.

Ja-Be-Ja



(a) A poor partitioning of a graph. Nodes are partitioned randomly so there are many inter-partition links.

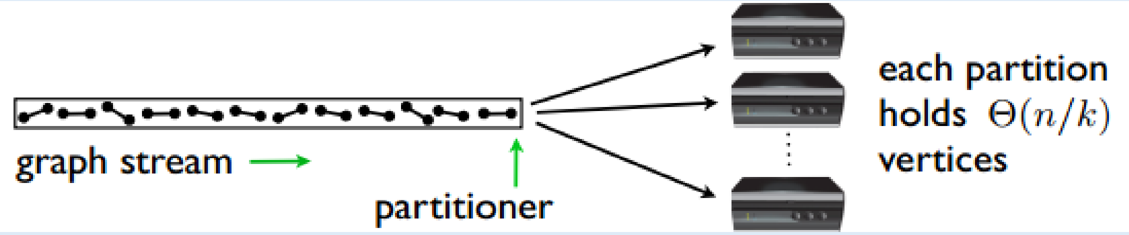
(b) A good partitioning of the same graph, where nodes that are highly connected are assigned to the same partition.

- *Balanced k-way graph partitioning*
  - Partitioning $\mathbf{G} = (V, E)$ into k equal-sized partitions with an offset ε
  - Partition function $\pi: V \rightarrow \{1, \ldots, k\}$ where $\pi(p)$ shows the partition of vertex
  - $N_p(\mathrm{c}) = \{q \in N_v : \pi(q) = \mathrm{c}\}$ i.e. $x_p(\mathrm{c}) = |N_p(c)|$ is the number of neighbors of with partition c and x_p is the number of neighboring nodes
  - Energy of the graph: $E(\mathbf{G}, \pi) = \frac{1}{2} \Sigma_{p \in V} (x_p - x_p(\pi_p))$
  - $\pi^* = \mathrm{argmin}_{\pi} E(\mathbf{G}, \pi)$ s.t. $|V(\mathrm{c}_1)| = |V(\mathrm{c}_2)|, \forall\, \mathrm{c}_1, \mathrm{c}_2 \in \{1, \ldots, k\}$
- **IDEA**: Initialize partitions at random and apply a local search heuristic towards lower energy state (min-cut)
  - Energy of the system is defined as the number of nodes with different colors
  - Energy of a node is defined as the number of its neighbors with different partitions

F. Rahimian, et al. "Ja-be-ja: A distributed algorithm for balanced graph partitioning." SASO, 2013.

Fennel: Streaming *k*-way graph partitioning

- **Streaming partitioning** == One pass partitioning
  - In streaming graph partitioning vertices are arrived and the decision of placement has to be done on-the-fly



- **IDEA**: Greedy scheme
- *Send vertex v to partition that maximizes*
  - $P = (S_1, \ldots, S_k)$ where S is a subset of V vertices set
  - $|V| = n$, $|E| = m$
  - e(S, $V \setminus S$) is the cut-edge across the cut (S, $V \setminus S$)
  - Edge cardinality $|e(S_i, S_i)|$ (both ends)

$$\frac{\sum_{i=1}^{k} e(S_i, V \setminus S_i)}{m} + \frac{1}{k} \sum_{i=1}^{k} \left(\frac{|S_i|}{\frac{n}{k}} \right)^{\gamma}$$

Edges Vertices

C. Tsourakakis, et al. "Fennel: Streaming graph partitioning for massive scale graphs." WSDM, 2014.

Spinner, Ja-Be-Ja, Argo and Fennel Comparison

- **Spinner**
  - Cloud (Giraph)
  - Vertex-centric
  - Balanced (edge)
  - Undirected graphs
  - *Arbitrary partition sizes (Capacity)*
  - Edge-cut
  - Label propagation
- **Ja-Be-Ja**
  - Theoretic
  - Vertex-centric
  - Balanced (edge)
  - Weighted graphs
  - *Arbitrary partition sizes (Initialization)*
  - Edge-cut
  - Local search
- **Argo**
  - HPC (MPI)
  - (Vertex-centric)
  - Balanced (weights)
  - Weighted graphs
  - *Arbitrary partition sizes (Quota)*
  - Resource contention
  - Linear deterministic greedy
- **Fennel**
  - Big Data
  - (Vertex-centric)
  - Balanced (relaxation)
  - Undirected graphs
  - *Arbitrary partition sizes (\Gamma)*
  - Edge-cut
  - Greedy scheme

Fennel: Comparison with Spinner & Metis

- What is the difference between Fennel and others?

	Twitter k=2		Twitter k=4		Twitter k=8		Twitter k=16		Twitter k=32	
Approach	ϕ	ρ	ϕ	ρ	ϕ	ρ	ϕ	ρ	ϕ	ρ
Wang et al. [33]	0.61	1.30	0.36	1.63	0.23	2.19	0.15	2.63	0.11	1.87
Stanton et al. [29]	0.66	1.04	0.45	1.07	0.34	1.10	0.24	1.13	0.20	1.15
Fennel [30]	0.93	1.10	0.71	1.10	0.52	1.10	0.41	1.10	0.33	1.10
Metis [18]	0.88	1.02	0.76	1.03	0.64	1.03	0.46	1.03	0.37	1.03
Spinner	0.85	1.05	0.69	1.02	0.51	1.05	0.39	1.04	0.31	1.04

$$\phi = \frac{\#\ local\ edges}{|E|}$$

$$\rho = \frac{maximum\ load}{\frac{|E|}{k}}$$

C. Tsourakakis, et al. "Fennel: Streaming graph partitioning for massive scale graphs." WSDM, 2014.

Discussion Outline

Graph Partitioning

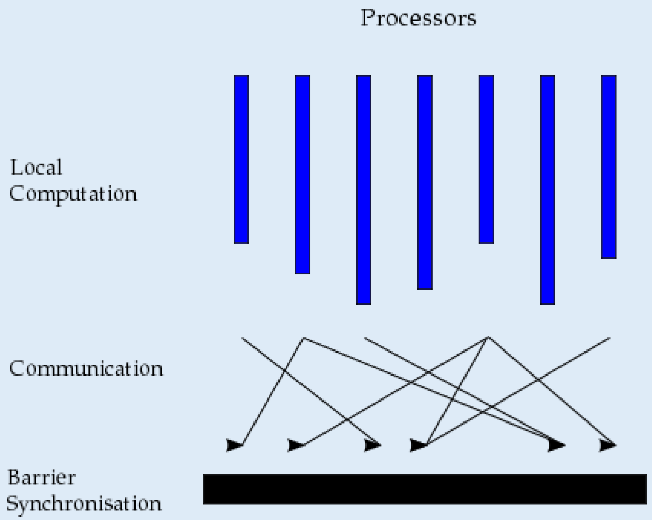
Vertex-centric, architecture-aware and streaming

Cloud-based Graph Analytics Platforms

- *Vertex-centric* (**GraphLab**, **Distributed GraphLab** and PowerGraph)
- Linear algebra

HPC-based Graph Analytics Platforms

Pregel: A Legacy Graph Processing Platform



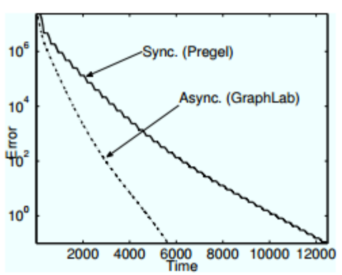
- Pregel and its open-source implementation Giraph
  - Bulk Synchrnous Processing (BSP)
  - Super-step
  - Vertex centric
  - Combiners (Aggregators)
- What makes a graph processing engine?
  - A **sequential code** that is executed concurrently on all vertices/edges.
  - The **engine** itself which is *iteratively* process the graph by running the vertices/edges code

G. Malewicz, et al. "Pregel: a system for large-scale graph processing." International Conference on Management of data, 2010.

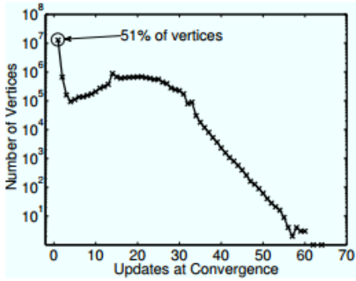
GraphLab: Machine Learning and Data Mining (MLDM) algorithm properties

- MapReduce limitations:
  - MapReduce fails when there is *computational dependencies*
  - MapReduce imposes a massive amount of I/O for iterative computations
  - MapReduce does not support *iterative workflow*
- MLDM requirments

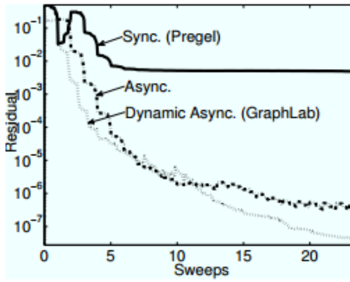
1. MLDM algorithms have *graph structured computation* (Dependent computation)
2. Asynchronous systems provide algorithmic benefits for MLDM **(a)** (Utilizing most recent data, avoiding stragglers effects and execution time variability)
3. Dynamic computation (Asymmetric convergence **(b)** and dynamic scheduling **(c)**)
4. Serializability: Ensuring parallel execution have an equivalent sequential execution **(d)**



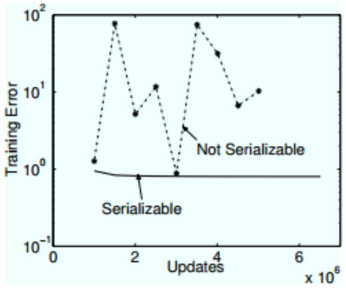
(a) Async vs Sync PageRank



(b) Dynamic PageRank



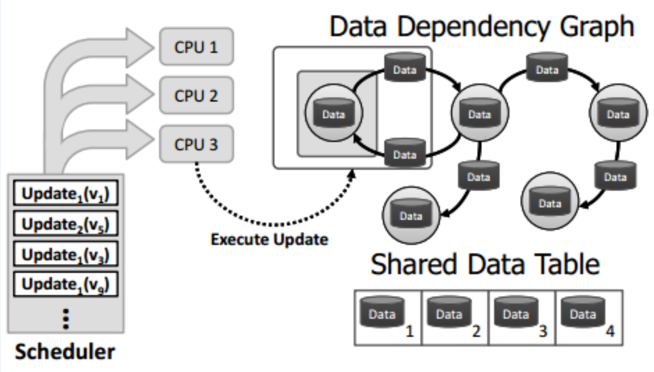
(c) LoopyBP Conv.



(d) ALS Consistency

Y. Low et al. “GraphLab: A New Framework For Parallel Machine Learning.” arXiv, 2014.
Y. Low et al. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud.“ VLDB, 2012.

GraphLab: Abstraction



- **Data Model**: GraphLab's low level abstraction (like MPI and Pthreads abstractions)
  - **Data graph**: **G** = (V, E) for representing program states
  - **Shared Data Table (SDT)**: **T**[key] → Value to support global shared state
- User defined computation
  - **Update function** (Map): Local computations

    $D_{Sv} \leftarrow f(D_{Sv}, \mathbf{T}) = f(v)$ where S_v is the neighborhood of v

    say S_v as **scope of *v***
  - **Synch mechanism** (Reduce): Global aggregations

    $r_{\mathrm{k}}^{(i+1)} = \mathrm{Fold}_k(D_v, r_k^{(i)})$ Aggregate data

    $r_k^{l} = \mathrm{Merge}(r_k^{i}, r_k^{j})$ If provided, parallel tree reduction is used

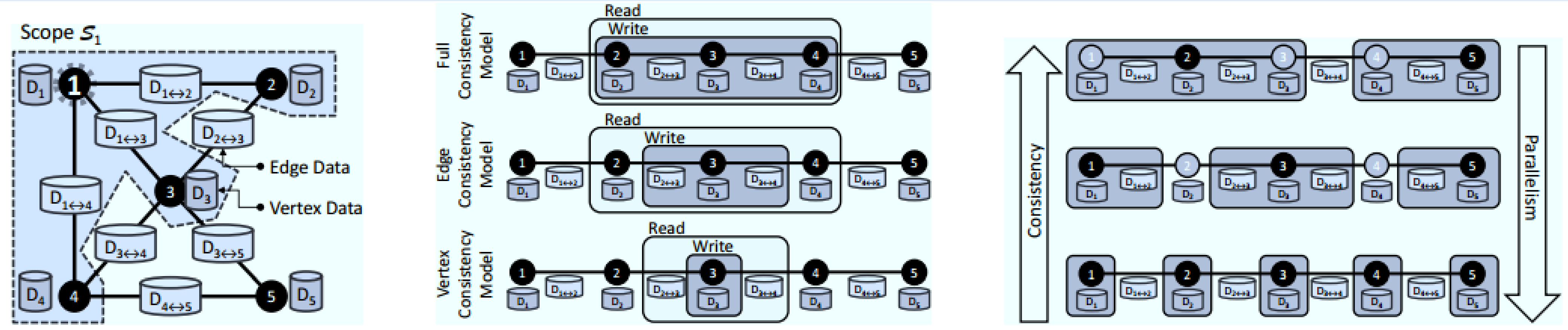
    $\mathbf{T}[k] = \mathrm{Apply}_k(r_k^{(|V|)})$ Write results
    - Unlike Pregel and Giraph, Synch runs continuously in the background
  - **Execution Model**: Starts with initial set T, removes vertices from T (`RemoveNext`(T)) and add new vertices back into T

Y. Low et al. "GraphLab: A New Framework For Parallel Machine Learning." arXiv, 2014.

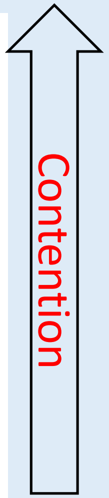
GraphLab: Consistency Model

- **Ensuring serializability**: *Full, edge and vertex consistency models* allow the runtime to optimize parallel execution while maintaining serializbility.
- The simultaneous execution of two update functions in overlapping scopes can lead to race-condition.

- Full consistency
  - Full read/write access in the scope
  - Scopes cannot have overlaps
- Edge consistency
  - Read/write access on the vertex and adjacent edges but only read to adjacent vertices
  - Slightly overlapping scopes
- Vertex consistency
  - Write access to the vertex read access to adjacent edges and vertices
  - All vertices can run update simultaneously



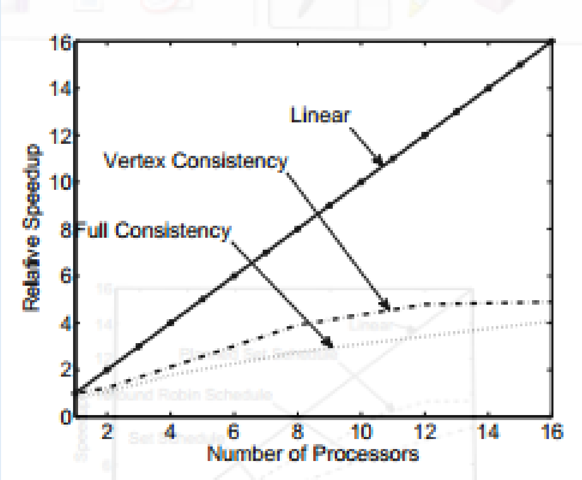
(a) Data Graph (b) Consistency Models (c) Consistency and Parallelism



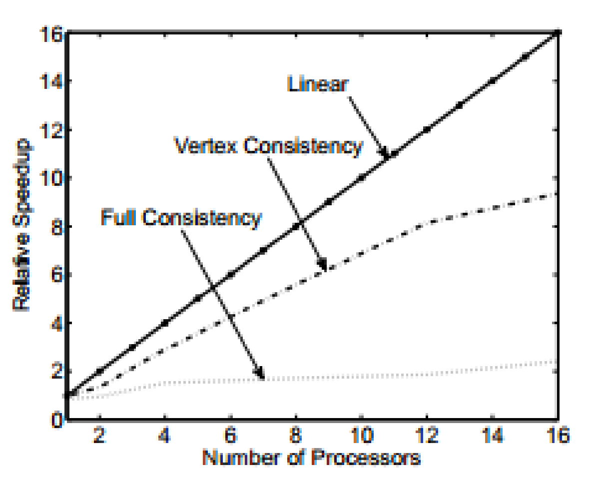
Y. Low et al. "GraphLab: A New Framework For Parallel Machine Learning." arXiv, 2014.

GraphLab: Consistency results

Shooting algorithm, sparse

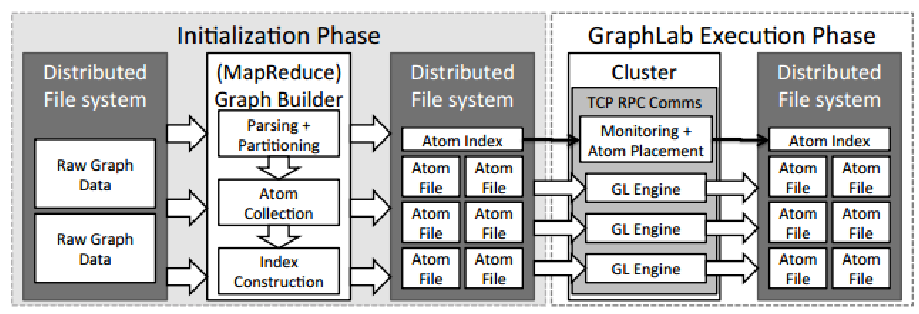


Shooting algorithm, dense



Y. Low et al. “GraphLab: A New Framework For Parallel Machine Learning.” arXiv, 2014.

Distributed GraphLab: Design



- **Two stage partitioning**
  - Graph is partitioned into *k atoms* (partitions) (*k* > number of machines)
    - **Ghost**: Set of vertices and edges adjacent to partition boundary. Serves the purpose of cache coherency
  - *Atom index* (a meta graph of *k* atoms) is partitioned among *machines*

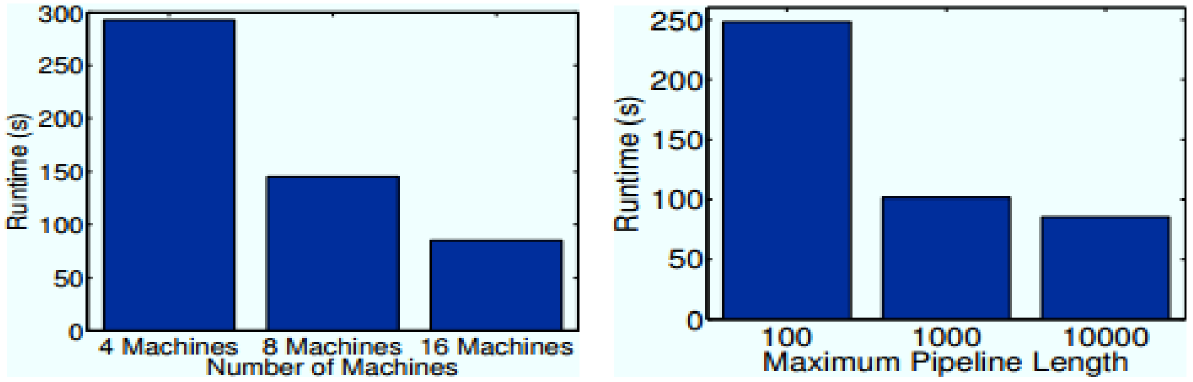
Execution engines

- Chromatic engine (Partially asynchronous):
  - *Edge and full consistencies* implemented using 1st and 2nd order vertex coloring to achieve serializable parallel execution
  - Hard to schedule, and availability of graph coloring prior to computation
- Distributed locking engine (asynchronous)
  - Associating a readers-writer lock with each vertex
  - *Vertex consistency* is achieved by acquiring a write lock on the central vertex of each scope
  - *Edge consistency* is achieved by acquiring a write lock on the central vertex and read locks on adjacent vertices
  - *Full consistency* is achieved by acquiring write locks on the central vertex and all adjacent vertices.
  - *Deadlocks* are voided using a canonical order: (machine ID, vertex ID(owner(*v*), *v*))

Y. Low et al. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud." VLDB, 2012.

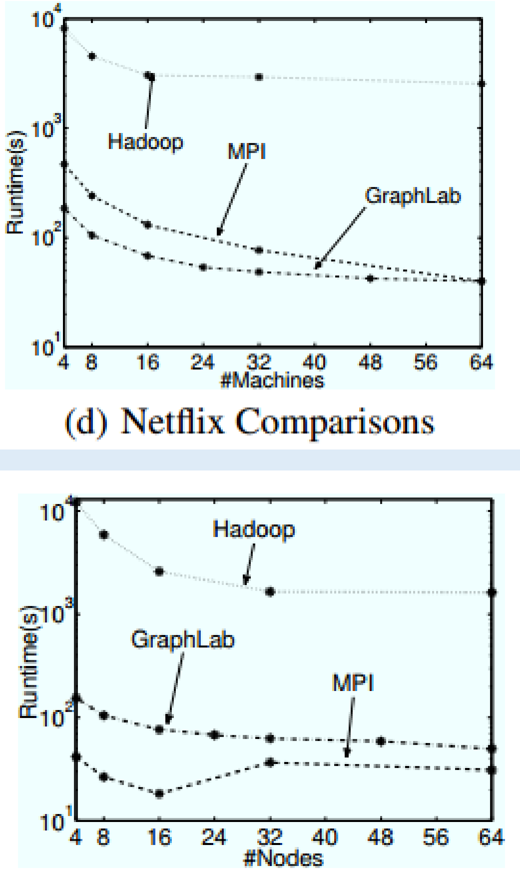
Distributed GraphLab: Results

Latency of remote lock acquisition and data synchronization



(a) Runtime (b) Pipeline Length

Netflix challenge



(d) Netflix Comparisons

(c) NER Comparisons

- Named Entity Recognition (NER)
  - The task of determining the type of a noun-phrase (e.g. a person) from its context
  - Poor computation to communication ratio
    - Computation ↓
    - Communication ↑

Y. Low et al. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud." VLDB, 2012.

Discussion Outline

Graph Partitioning

Vertex-centric, architecture-aware and streaming

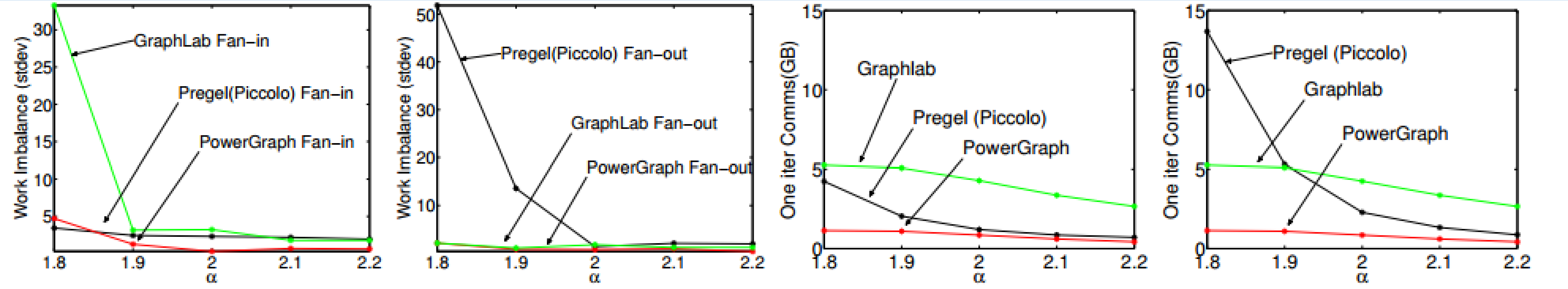
Cloud-based Graph Analytics Platforms

- *Vertex-centric* (GraphLab, Distributed GraphLab and **PowerGraph**)
- Linear algebra

HPC-based Graph Analytics Platforms

PowerGraph: Challenges of Natural Graphs

- Natural graphs have the properties of *skewed power-law degree distribution*.
  - a small fraction of the vertices are adjacent to a large fraction of the edges.
    - E.g. celebrities in a social network.
    - 1% of the vertices in the Twitter graph are adjacent to nearly 50% the edges.
- Under *power-law degree distribution* the probability that a vertex has degree d is $\mathbf{P}(d) \propto d^{-\alpha}$ i.e. $\alpha > 0$ controls the skewness
  - Natural graphs have a power-law constant $\alpha \sim 2$
  - Internet has a power-law constant $\alpha \sim 2.2$



(a) Power-law Fan-In Balance (b) Power-law Fan-Out Balance (c) Power-law Fan-In Comm. (d) Power-law Fan-Out Comm.

J. E. Gonzalez, et al. "PowerGraph: Distributed graph-parallel computation on natural graphs." OSDI, 2012.

PowerGraph: Abstraction – Gather, Apply and Scatter (GAS) Model

- **Gather**: $\Sigma \leftarrow \oplus_{v \in \mathbf{N}(u)} g(D_u, D_{(u,v)}, D_v)$ (Fan-in)
  - Collect information from adjacent edges
  - Commutative and associative
- **Apply**: $D_u^{new} \leftarrow a(D_u, \Sigma)$
  - Update the value of the central vertex
- **Scatter**: $\forall\, v \in \mathbf{N}(u)$: $(D_{(u,v)}) \leftarrow s(D_u^{new}, D_{(u,v)}, D_v)$ (Fan-out)
  - Update the data of adjacent vertices
- E.g. PageRank
  - Gather → in-edges, Scatter → out-edges

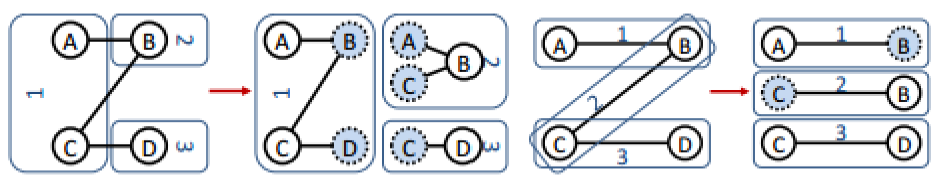
```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_(u,v), D_v) → (D_(u,v)^new, Accum)
}
```

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** Map
    $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{(u,v)}, D_v))$ Reduce
  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 $a_v \leftarrow$ Empty
end

J. E. Gonzalez, et al. "PowerGraph: Distributed graph-parallel computation on natural graphs." OSDI, 2012.

PowerGraph: Distributed Graph Placement



(a) Edge-Cut (b) Vertex-Cut

- *Percolation theory* suggests that power-law graphs have good vertex-cut.
  - Intuition: Cutting very high degree vertices into smaller fractions (i.e. $E >> V$)
- *Balanced p-way vertex cut*

$$\min_{A} \frac{1}{|V|}\sum_{v\in V}|A(v)| \quad \text{s.t.} \quad \max_{m} |\{e \in E \mid A(e) = m\}|, < \lambda \frac{|E|}{p}$$

Number of replicas Uniform dis. of edges

  - *Vertices* can span over multiple machines
  - Each vertex can have multiple *replicas* (*master*, *mirrors*)
    - $A(v)$ is the set of machines have a replica of vertex v
  - *Edges* are assigned to machines evenly and stored only once
  - Two implementations
    - *Randomized vertex-cut* for p machines
    - *Greedy vertex-cut for edge* (u, v)
      - Coordinated, Oblivious

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

J. E. Gonzalez, et al. "PowerGraph: Distributed graph-parallel computation on natural graphs." OSDI, 2012.

PowerGraph: Distributed Graph Placement (continued)

- Balanced *p*-way vertex cut

1. *Randomized vertex-cut* for *p* machines
    - The simplest way to have a vertex cut is to randomly assign vertices to machines
    - Then uses balanced vertex-cut objective to balance edges

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

2. *Greedy vertex-cut for edge* (u, v)
    - placing the i+1 edge (u, v) after having placed the previous i edges
        - $A(u) \cap A(v)$ → Assign e_{i+1} to the intersection machine
        - $((A(u) \cap A(v)) = \emptyset) \wedge (A(v) \neq \emptyset \cap A(v) \neq \emptyset)$ → Assign e_{i+1} to the machine with less edges
        - $((A(u) = \emptyset) \wedge (A(v) \neq \emptyset)) \vee ((A(u) \neq \emptyset) \wedge (A(v) = \emptyset))$ → Assign e_{i+1} to the available machine
        - $((A(u) = \emptyset) \wedge (A(v) = \emptyset))$ → Assign e_{i+1} to the least loaded machine

J. E. Gonzalez, et al. "PowerGraph: Distributed graph-parallel computation on natural graphs." OSDI, 2012.

Discussion Outline

Graph Partitioning

Vertex-centric, architecture-aware and streaming

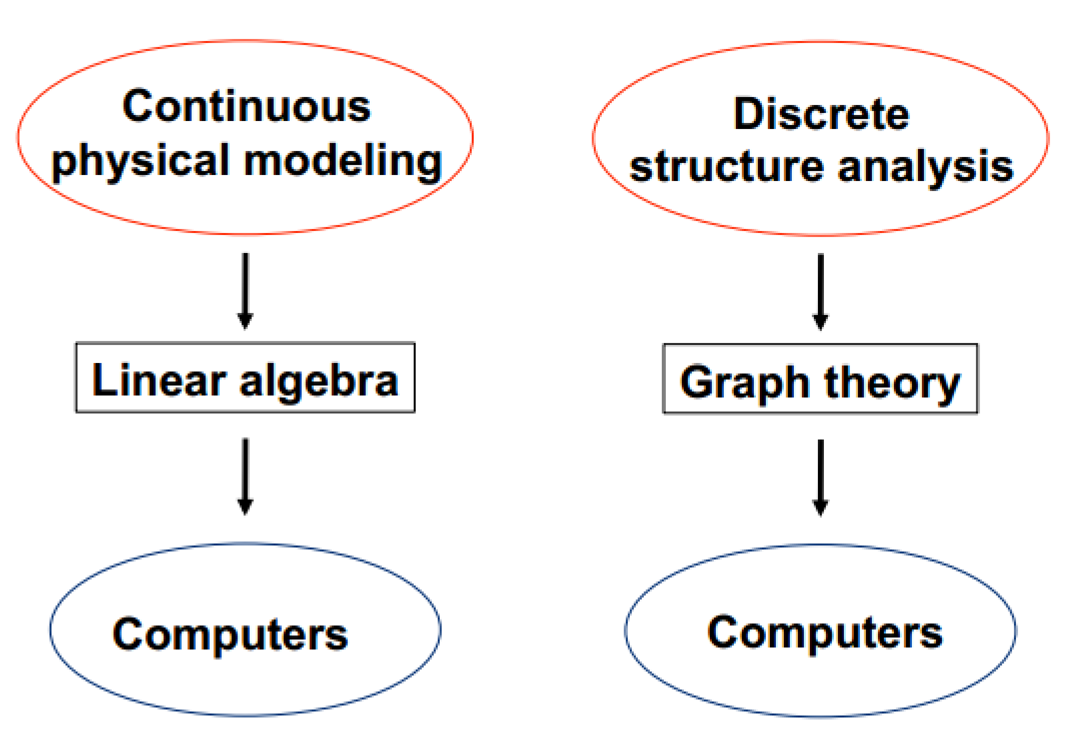
Cloud-based Graph Analytics Platforms

- Vertex-centric (GraphLab, Distributed GraphLab and PowerGraph)
- *Linear Algebra* (**LA3**)

HPC-based Graph Analytics Platforms

Linear Algebra as an Alternate for Graph Theory

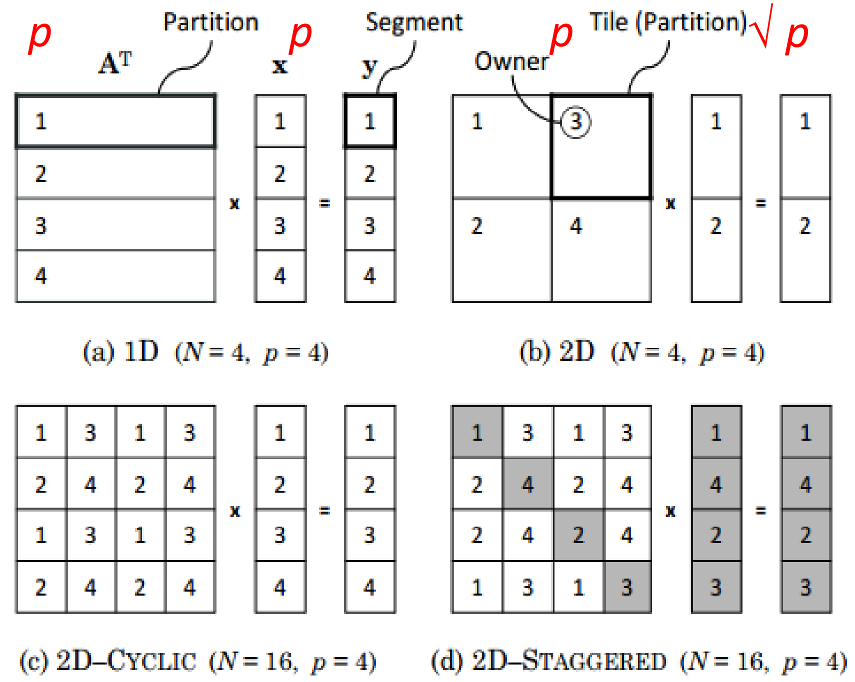
- Large combinatorial graphs appears in
  - Computational biology, analytics, web search, dynamic systems, and sparse matrix methods
- Leveraging the duality between graphs and sparse matrices
  - Adjacency matrix is considered as a sparse matrix data structure
  - Linear algebra primitives on this matrix map to certain graph operations
    - **SpMV**: $y = A \times x$
    - **SpMM**: $C = A \times B$



A. Buluç, et al. "The Combinatorial BLAS: Design, implementation, and applications." The International Journal of High Performance Computing Applications, 2011.
D. Bader, et al. "The Graph BLAS effort and its implications for Exascale." SIAM, 2014.

LA3: Design

- Programming model (Initi, Scatter, Gather, Combine, Apply)
- Pre-processing
  1. Vertex classification:
     - *Regular*, *source*, *sink*, and *isolated*
     - Row-group, and column-group. Group leader for classifying vertices
  2. Edge processing:
     - Each tile is spitted into sub-tiles
     - Increasing cache/memory locality
- Partitioning (Tile, Segment)
  - 1-D partitioning (Edge-cut): Imbalanced tiles due to skewness
  - 2-D partitioning (Vertex-cut): Imbalanced tiles due to skewness
  - 2-D Cyclic and 2-D Staggered: Higher parallelism, more balanced
- Execution Engine
  - Computation filtering (Pre-loop, Main-loop and Post-loop)
  - Communication filtering (Eliminating communication for empty tiles)
  - Pseudo-asynchronous Computation and Communication (2D-STAGGERED)



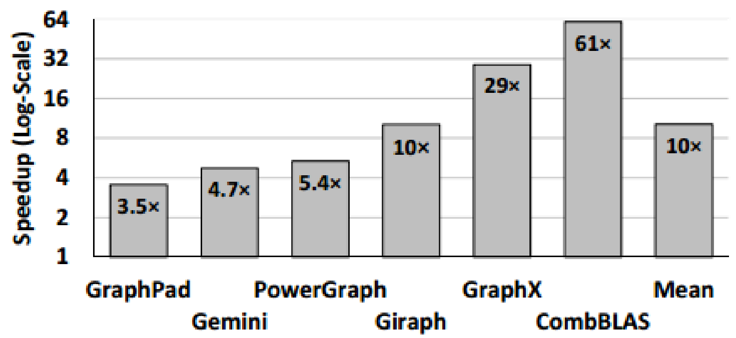


Figure 1: LA3's speedup versus other systems averaged over various standard applications and datasets. Mean speedup is 10× over all systems.

Y. Ahmad, et al. "LA3: A Scalable Link- and Locality-Aware Linear Algebra-Based Graph Analytics System" VLDB, 2018

Discussion Outline

Graph Partitioning

Vertex-centric, architecture-aware and streaming

Cloud-based Graph Analytics Platforms

Vertex-centric, *Linear algebra*

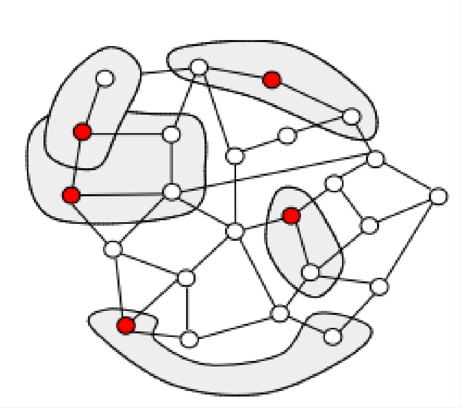
HPC-based Graph Analytics Platforms

- *NUMA-aware* (**Galios**, Gemini and Mosaic)

Galois: Amorphous Data Parallelism (ADP) Programming Model

1. Active nodes (red dots)
    - When?
        - Autonomous scheduling (worklist): More parallelism, high diameter graphs
        - Coordinated scheduling (BSP): Less parallelism, low diameter graphs
2. Neighborhood (gray clouds)
3. Operator: Morph the graph by adding or removing active nodes
    - Push style: Reads from active node and writes to its neighbors
    - Pull style: Reads from its neighbors and writes to the active node
        - Requires less synchronization

- Galois borrowed two concepts fro OS:
    - 1. Priority scheduling, 2. Memory Allocator
    - Typical tasks in graph processing take only microseconds to execute

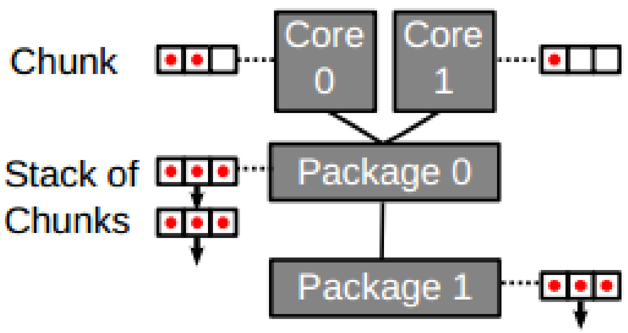


	Cycles	Inst.
bfs	6007	2077
sssp	1521	308
dia	7265	2296
cc	5063	1380
pr	3190	541

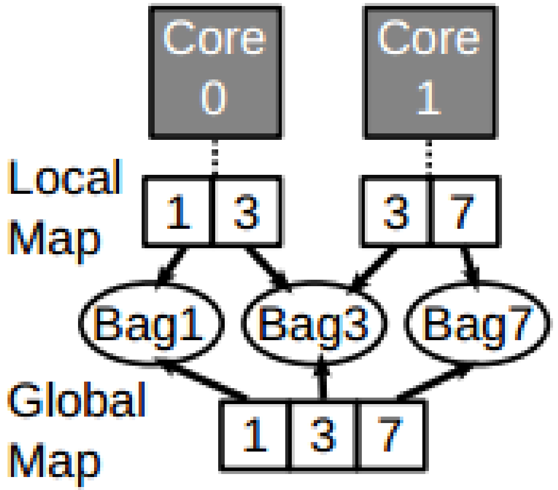
D. Nguyen, et al. "A lightweight infrastructure for graph analytics." SOSP, 2013.

Galois: Schedulers

- **Basic scheduler**: Topology aware bag of tasks (vertices)
  - **Chunk** (→ Core): *8 – 64 vertices* processing with LIFO policy
    - **Package (→ Sockets)**: *A list of chunks* processed with LIFO policy
      - **Bag**: A list of packages
  - When chunks associated with a core becomes empty, it is moved to the package-level list
    - If package-level list is empty, the core probes other packages
      - One core is always responsible for probing package-level list for hungry cores.
- **Obim scheduler**: A priority scheduler with *a sequence of bags*.
  - *Each **bag** is associated with a **priority** level*
  - **Global Map**: A sparse global data structure for locating tasks by threads
  - **Local Map**: A lazy cache portion of the global map known to the thread.
  - *Global/local maps operations*:
    - **Updating the map** is done via a *global log*
    - **Pushing a task** via creating a new mapping in the global map
    - **Retrieving a task** only when the bag a thread is working on becomes empty
      - *Back-scan*: Scanning the global map for earlier priorities.



(a) A bag.



(b) Map in obim.

D. Nguyen, et al. "A lightweight infrastructure for graph analytics." SOSP, 2013.

Galois: Memory Allocator

- Memory allocator: A scalable multi-threaded algorithm that directly addresses NUMA concerns
  - **A slab allocator** for allocations in the runtime
    - A central page pool of huge pages
      - The page pool is *NUMA-aware* and can be *reclaimed*
      - Each application preallocates some number of pages prior to execution
    - Separate allocators for each block size
    - Each thread maintains a free list of blocks
      - If empty, a bump-pointer region allocator is used to divide the page into blocks
  - **A Bump-pointer region allocator** for allocations from user code
    - Used for variable-sized allocations required by temporaries created by user code
    - If the allocation size exceeds page size (2 MB), the allocator falls back to `malloc`

D. Nguyen, et al. "A lightweight infrastructure for graph analytics." SOSP, 2013.

Galois: NUMA-aware Optimizations

- **Topology-aware synchronization**:
  - The most common synchronization is among cores on the same package (**socket**) that share the same L3 cache
    - Threads in a package communicate via a shared counter
    - Much faster compared to Pthread barriers
- **Code size optimizations**:
  - **Reduce the runtime cost** of features by having a specialized implementation of an operator which is generated at compile time and only supports the required features.
    - Checking new tasks requires 4 instructions (a load, a branch, and 2 stores), on average this is 2% of SSSP instructions.
    - Tight loops are more likely to fit in L1 instruction cache

D. Nguyen, et al. "A lightweight infrastructure for graph analytics." SOSP, 2013.

Discussion Outline

Graph Partitioning

Vertex-centric, architecture-aware and streaming

Cloud-based Graph Analytics Platforms

Vertex-centric, *Linear algebra*

HPC-based Graph Analytics Platforms

- *NUMA-aware* (Galios, **Gemini** and Mosaic)

Gemini: Motivation

Computation rather than communication appears to be the actual bottleneck of distributed systems

1. We lose system efficiency as we move from single-thread to shared memory, then to distributed implementations.
2. Active vertices are changing:
   - E.g. CC: Dense → Spare, SSSP: Sparse → dense → Sparse
3. Active vertices requires different communication patterns
   - Sparse edge set: Push model →
   - Dense edge set: Pull model ←

- Gemini extends **Ligra** to distributed systems
  - Adaptive switch between sparse and dense representations according to threshold $|E|/20$ in a shared memory machine.
- Gemini borrows the concept of master/mirror vertices from **PowerGraph** where graph is partitioned and vertices are distributed across different nodes
  - Sparse (push) mode: Master → Mirrors
  - Dense (pull) mode: Mirrors → Master
  - 1 message per active *master-mirror* pair (O(E) → O(V) messages)

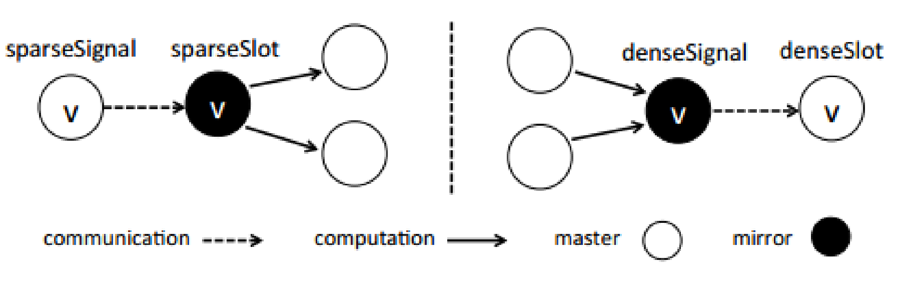
Shared memory

Single thread

Distributed

Cores	1	24 × 1		24 × 8	
System	OST	Ligra	Galois	PowerG.	PowerL.
Runtime (s)	99.9	21.9	19.3	40.3	26.9
Instructions	525G	496G	482G	7.15T	6.06T
Mem. Ref.	15.8G	32.3G	23.4G	95.8G	87.2G
Comm. (GB)	-	-	-	115	38.1
IPC	1.71	0.408	0.414	0.500	0.655
LLC Miss	8.77%	43.9%	49.7%	71.0%	54.9%
CPU Util.	100%	91.7%	96.8%	65.5%	68.4%

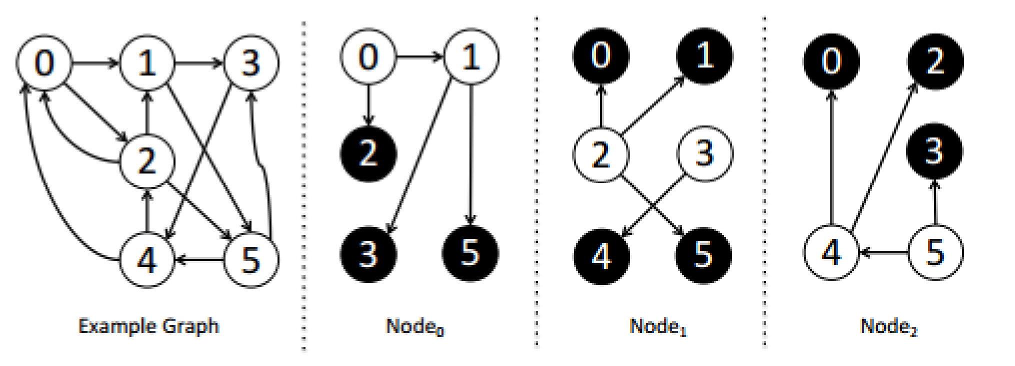
220 iterations of PageRank on Twitter



X. Zhu, et al. "Gemini: A Computation-Centric Distributed Graph Processing System." OSDI. 2016.
J. Shun, et al. "Ligra: a lightweight graph processing framework for shared memory." PPoPP, 2013.

Gemini: 2 Level Chunk-based Partitioning

1. ***Partitions vertices into contiguous chunks to preserve locality***
   - **Vertices** of a p-node cluster **G** is partitioned into p contiguous vertex chunks $(V_0, .., V_{p-1})$
     - E.g. Facebook friendship or, Geo-locations are closed together
     - Scalable when having random accesses
     - Sacrifice balanced edge distribution to some degree
     - Contiguous memory pages, thus reducing the memory footprint and preserving locality (Is it TRUE in practice?)
   - **Edges** are balanced by:
     - $\alpha\,|V_i| + |E_i^D|$ s.t. $\alpha = 8(p-1)$
     - $E_i^S = \{(src, dst, value) \in E \mid dst \in V_i\}$
     - $E_i^D = \{(src, dst, value) \in E \mid src \in V_i\}$



X. Zhu, et al. "Gemini: A Computation-Centric Distributed Graph Processing System." OSDI. 2016.

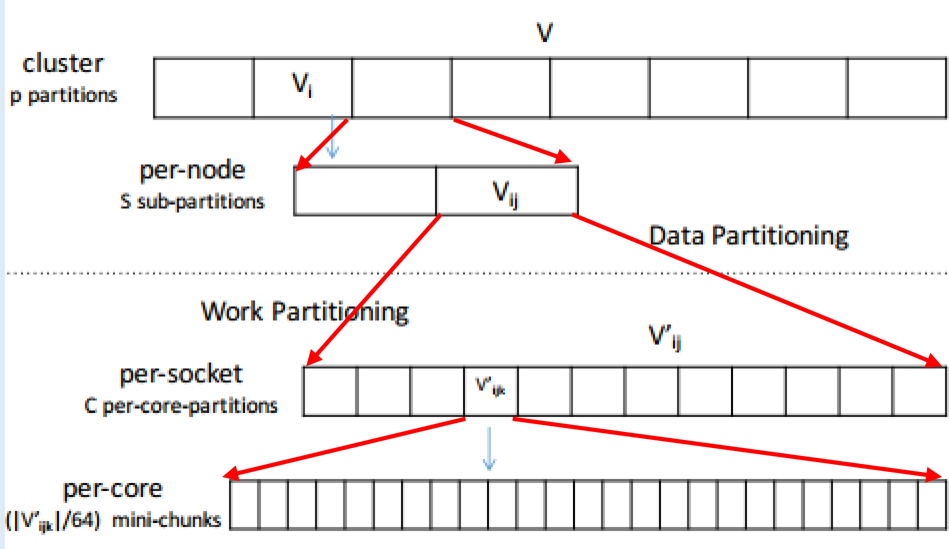
Gemini: 2 Level Distributed Graph Representation (continued)

2. NUMA-aware sub-partitioning per node with s sockets

- Continues chunks → sub-chunks of size V_i/s
- Improving both sequential and random accesses
- Faster memory access and better utilization of LLC
- Avoid remote access to other sockets

- Multi-level chunk-based partitioning
  - Sub-chunks → *per-core chunks* of size 64 vertices
- Task scheduling: Threads can steal mini-chunks from others (interleaved chunks)

Graph (cluster)
→ Chunks (nodes)
→ Sub-chunks (sockets)
→ Per-core chunks (cores)
→ mini chunks of 64 vertices



X. Zhu, et al. "Gemini: A Computation-Centric Distributed Graph Processing System." OSDI. 2016.

Gemini: Results

10x 2x

Graph	Raw	PowerGraph	Gemini
enwiki-2013	0.755	13.1	4.02
twitter-2010	10.9	138	32.1
uk-2007-05	27.8	322	73.1
weibo-2013	47.9	561	97.5
clueweb-12	318	-	597

Table 5: Peak 8-node memory consumption (in GB). "-" indicates incompletion due to running out of memory.

$p \cdot s$	T_{PR} (s)	$\Sigma\|V_i\|/(p \cdot s)$	$\Sigma\|E_i\|/(p \cdot s)$	$\Sigma\|V_i'\|/(p \cdot s)$
$1 \cdot 2$	12.7	20.8M	734M	27.6M
$2 \cdot 2$	7.01	10.4M	367M	19.6M
$4 \cdot 2$	$\boxed{3.88}$	5.21M	184M	$\boxed{13.5M}$
$8 \cdot 2$	$\boxed{3.02}$	2.60M	91.8M	$\boxed{10.5M}$

Table 6: Subgraph sizes with growing cluster size

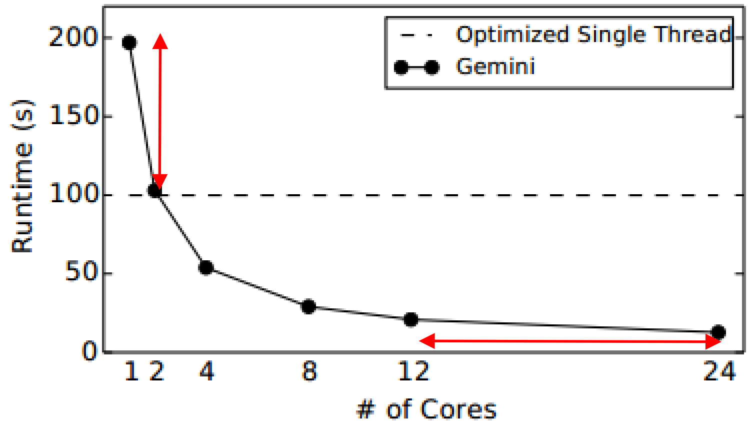


Figure 9: Intra-node scalability (*PR* on *twitter-2010*)

X. Zhu, et al. "Gemini: A Computation-Centric Distributed Graph Processing System." OSDI. 2016.

Discussion Outline

Graph Partitioning

Vertex-centric, architecture-aware and streaming

Cloud-based Graph Analytics Platforms

Vertex-centric, *Linear algebra*

HPC-based Graph Analytics Platforms

- *NUMA-aware* (Galios, Gemini and **Mosaic**)

Mosaic: Processing a Trillion Edges Graph on a Single Machine

- **Trillion Edges Challenge**:
  - Facebook largest graph has 1.4 billion vertices and 1 trillion edges.
  - Giraph requires 200 nodes for processing it.
- **Hardware specifications**:
  - Host processor: Non-uniform Memory Access(NUMA) architecture
    - 2 sockets, 12 cores each
  - Coprocessor (A supercomputer on card): 4 Xeon Phi with 61 cores each with
    - 4 hardware threads
    - 512-bit SIMD unit
    - 1.224 GHz speed
    - 512KB L2 cache
  - 6 NVMe SSD (1.2 TB): Allows terabytes of storage with up to 10x throughput than SSDs
  - RAM: 768 GB

- **Implementation**: 17 K lines of code in C++
- **Dividing components of a graph processing**:
  - **Scale-up**: Memory intensive operations, e.g. *vertex-centric operations* are offloaded to fast *host processors*
  - **Scale-out**: Compute and I/O intensive operations, e.g. *edge-centric operations* are offloaded to *coprocessors*

S. Maass, et al. "Mosaic: Processing a trillion-edge graph on a single machine." EuroSys, 2017

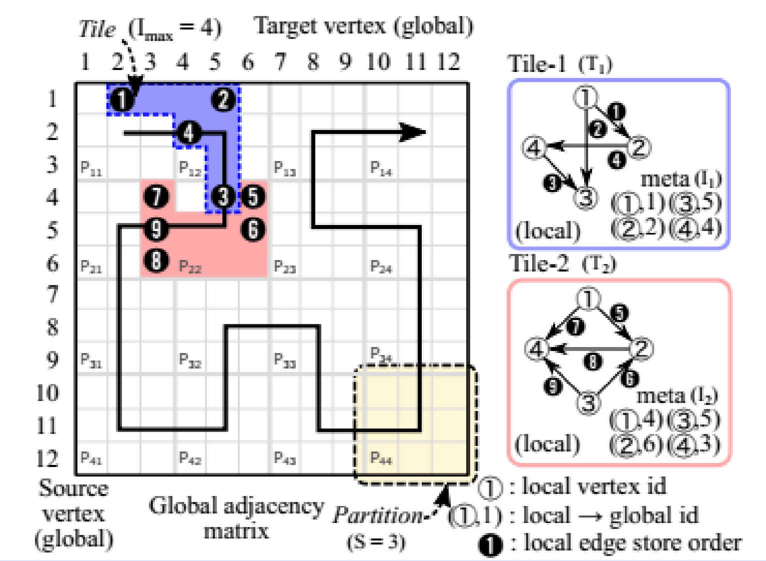
Mosaic: Tiles - Local Graph Processing Units

- Graph data structure
  - Depending on size of the graph, vertices are identified a by 32-bit or 64-bit integer (4–8 bytes)
  - Tiles (subgraphs) data structures
    1. Each tile is an independent unit of edge processing
    2. Tiles are evenly distributed among coprocessors
- Inside a Tile
  - The number of unique vertices in a tile is bounded by I_{max}
  - The number of edges per tile varies (Static load balancing)
  - Tiles are of size S x S i.e. $S = 2^{16}$
  - $I_{max}=2^{16}$ and Integer vertex IDs, *per tile storage* is $2^{16} * 4$ bytes = 256 KB < 512 KB L2 cache size

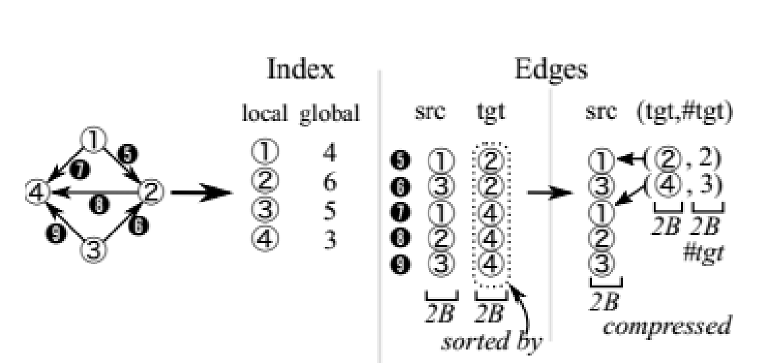
S. Maass, et al. "Mosaic: Processing a trillion-edge graph on a single machine." EuroSys, 2017

Mosaic: Tiles - Local Graph Processing Units (Continued)

- On-disk data structure:
  - Tile index: local → global
  - Edges:
    - Edge list
    - CSR (#target vertices > 2 * #edges)
  - Locality:
    - Sequential accesses to the edges in local graph
    - Write locality by storing edges in sorted order
- Conversion:
  - Stream of partitions of adjacency matrix of global graph → S x S i.e. $S = 2^{16}$
  - Edges are consumed following Hilbert-ordered with $I_{max} = 2^{16}$
- Hilbert-ordered tiling
  - Traversing tiles in a certain order (P_{11}, P_{12}, P_{22}, ..) P_{ij} → d
  - Preserving locality while traversing tiles
  - I/O prefetching



I_{max} = 4 and S = 3

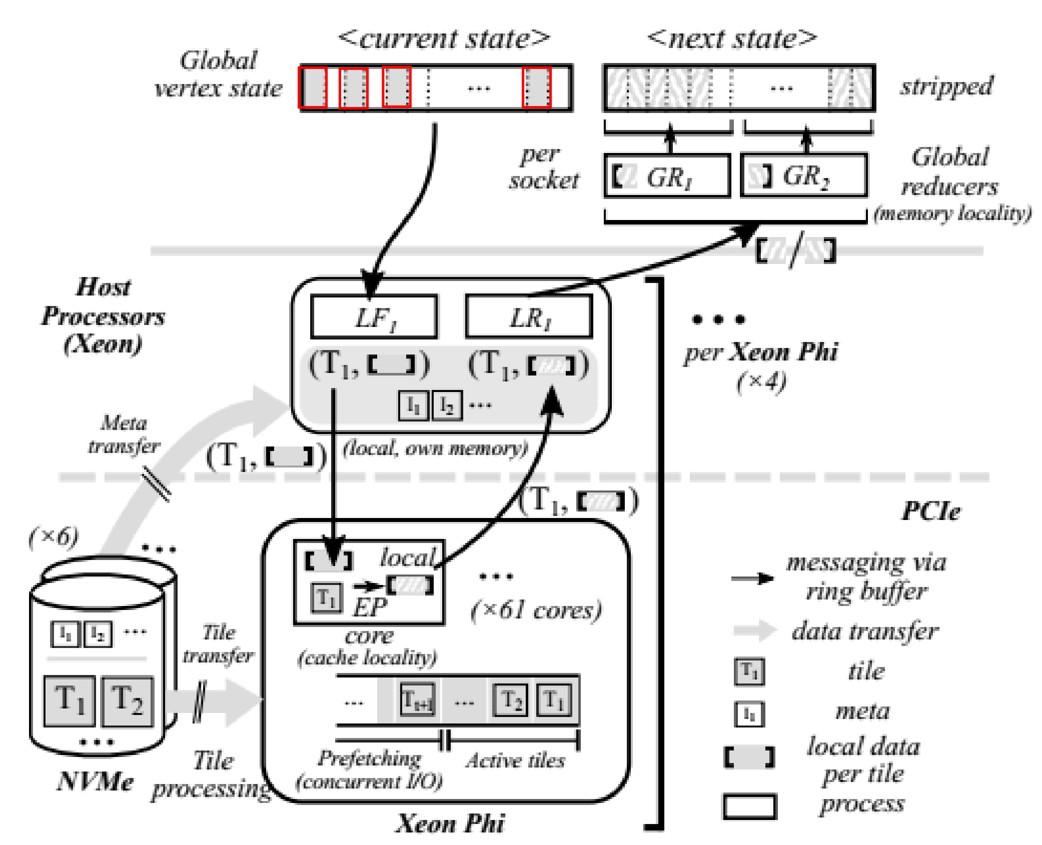


Reducing the number of bytes by 20%

S. Maass, et al. "Mosaic: Processing a trillion-edge graph on a single machine." EuroSys, 2017

Mosaic: System Components

- **Scale-out components** (Using pairs of Xeon Phis and NVMes)
  - *Local Fetcher*: Given a tile extracts the vertices
  - *Edge Processor*: Given a set of vertices, extracts the edges from a tile, executes the algorithm on edges and send results to local reducer
  - *Local Reducer*: Aggregates vertices state and send to global reducer
- **Scale-up componenets** (Using host processors)
  - *Global Reducer*: Disjoint partitions of vertices are assigned to sockets responsible for receiving data from local reducer and updating vertices
  - *Striped partitions*: Stripes of vertices are interleaved among NUMA nodes



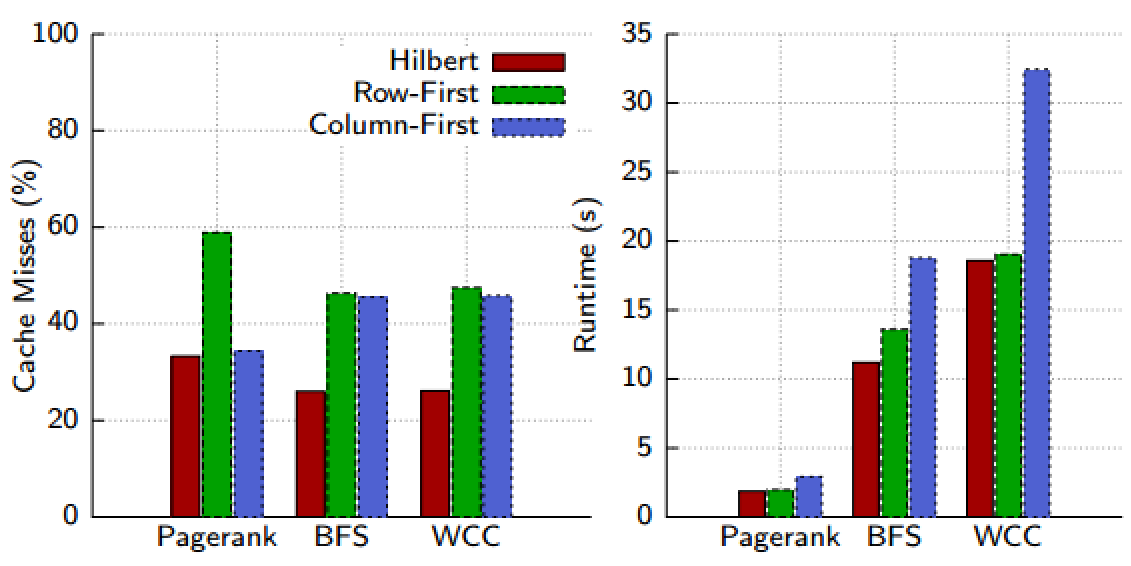
S. Maass, et al. "Mosaic: Processing a trillion-edge graph on a single machine." EuroSys, 2017

Mosaic: Results

Up to 68% reduction in data size

Graph	#vertices	#edges	Raw data	Mosaic size (red.)
*rmat24	16.8 M	0.3 B	2.0 GB	1.1 GB (−45.0%)
twitter	41.6 M	1.5 B	10.9 GB	7.7 GB (−29.4%)
*rmat27	134.2 M	2.1 B	16.0 GB	11.1 GB (−30.6%)
uk2007-05	105.8 M	3.7 B	27.9 GB	8.7 GB (−68.8%)
hyperlink14	1,724.6 M	64.4 B	480.0 GB	152.4 GB (−68.3%)
*rmat-trillion	4,294.9 M	1,000.0 B	8,000.0 GB	4,816.7 GB (−39.8%)

45% better cache locality and
up to 43% reduction in runtime



S. Maass, et al. "Mosaic: Processing a trillion-edge graph on a single machine." EuroSys, 2017

Summary

	In-memory	Out-of-core
Single machine	Galois GraphLab	Mosaic
Distributed	Pregel Giraph PowerGraph Dist. GraphLab LA3 Gemini	

Summary

	Synchronous	Asynchronous
Graph	Pregel Giraph Dist. GraphLab PowerGraph Gemini Mosaic	GraphLab Dist. GraphLab PowerGraph
SpMV		LA3

Summary

- Graph partitioning plays a crucial rule in balancing computation and computation across machines of a cluster.
- Graph processing engines are being built for certain applications
  - Machine learning and data mining
  - Linear algebra
  - Graph traversal
- These engines require optimizations in different layers
  - Hardware: NUMA-awareness, storage locality
  - Data distribution: partitioning
  - Network: Message passing
- Here, we survey a couple of engines and algorithms and investigate their characteristics.

References

- Graph Partitioning
  - Architecture aware
    - A. Zheng, et al. “Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing.“ Big Data, 2014.
    - A. Zheng, et al. “PARAGON: Parallel Architecture-Aware Graph Partition Refinement Algorithm .“ EDBT, 2016.
    - A. Zheng, et al. “Planar: Parallel Lightweight Architecture-Aware Adaptive Graph Repartitioning.“ ICDE, 2016.
    - A. Zheng, et al. “Argo: Architecture-aware graph partitioning.” Big Data, 2016.
  - Vertex-centric
    - C. Martella, et al. "Spinner: Scalable graph partitioning in the cloud." ICDE, 2017.
    - F. Rahimian, et al. "Ja-be-ja: A distributed algorithm for balanced graph partitioning." SASO, 2013.
  - Streaming
    - C. Tsourakakis, et al. "Fennel: Streaming graph partitioning for massive scale graphs." WSDM, 2014.

References

- Cloud-based Graph Analytics Platforms
  - Vertex-centric
    - Y. Low et al. “GraphLab: A New Framework For Parallel Machine Learning.” arXiv, 2014.
    - Y. Low et al. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud.“ VLDB, 2012.
    - J. E. Gonzalez, et al. "PowerGraph: Distributed graph-parallel computation on natural graphs." OSDI, 2012.
  - Linear Algebra Engines
    - A. Buluç, et al. "The Combinatorial BLAS: Design, implementation, and applications." The International Journal of High Performance Computing Applications, 2011.
    - D. Bader, et al. "The Graph BLAS effort and its implications for Exascale." SIAM, 2014.
    - Y. Ahmad, et al. "LA3: A Scalable Link- and Locality-Aware Linear Algebra-Based Graph Analytics System" VLDB, 2018
- HPC-based Graph Analytics Platforms
  - NUMA-aware
    - D. Nguyen, et al. "A lightweight infrastructure for graph analytics." SOSP, 2013.
    - X. Zhu, et al. "Gemini: A Computation-Centric Distributed Graph Processing System." OSDI, 2016.
    - S. Maass, et al. "Mosaic: Processing a trillion-edge graph on a single machine." EuroSys, 2017