Big Data Analytics

Lucas Rego Drumond

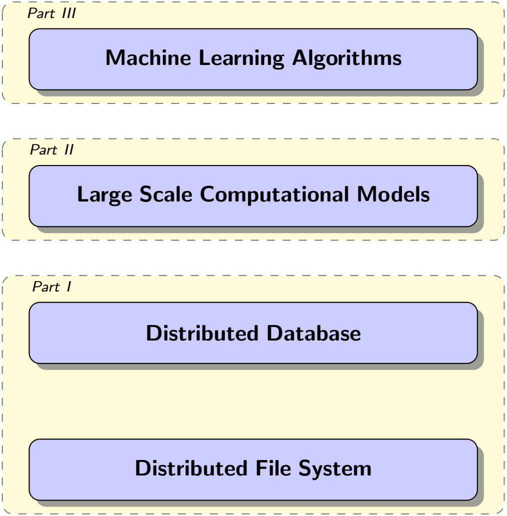
Information Systems and Machine Learning Lab (ISMLL)
Institute of Computer Science
University of Hildesheim, Germany

GraphLab

Outline

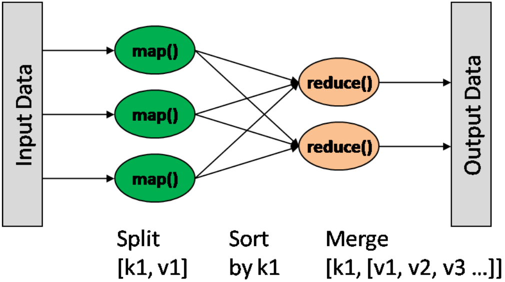
Outline

Overview

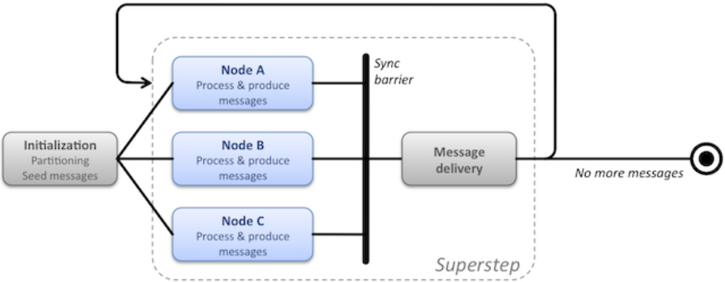

Part III
Machine Learning Algorithms
Part II
Large Scale Computational Models
Part I
Distributed Database
Distributed File System

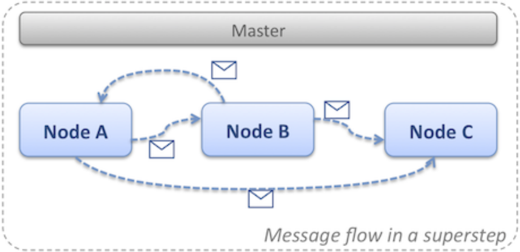
MapReduce - Review

1. Each mapper transforms a set key-value pairs into a list of output keys and intermediate value pairs
2. all intermediate values are grouped according to their output keys
3. each reducer receives all the intermediate values associated with a given keys
4. each reducer associates one final value to each key



Pregel - Review





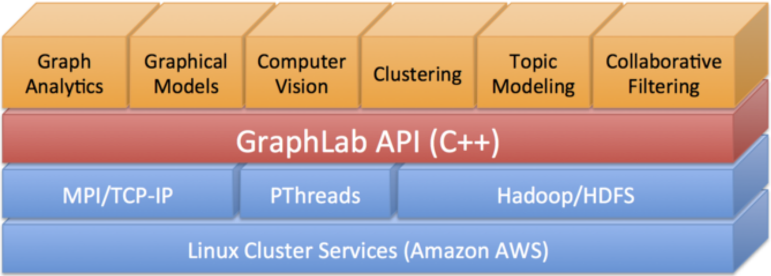
Outline

GraphLab

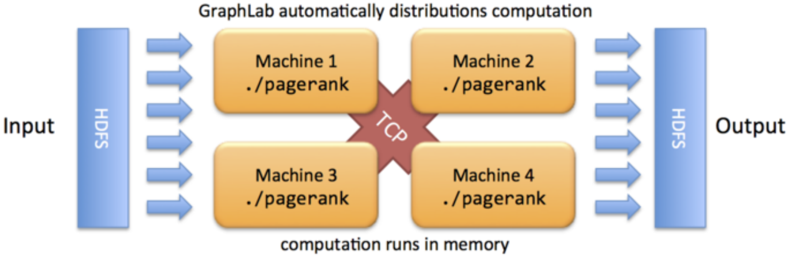
- A new framework for parallel machine learning
- Represents data as a directed graph
- Blocks of data stored in vertices and edges
- Globally shared data table
- Shared memory (GraphLab 1.0) distributed (GraphLab 2.1)

Available for download at: `http://graphlab.org`

GraphLab Software Stack



GraphLab System Overview



GraphLab Abstraction

The **GraphLab abstraction** consists of the following main components:

- A **Data Graph** provides a high level representation for data and complex computational dependencies
- Configurable **consistency models** for automatically guaranteeing data consistency
- **Scheduling Primitives** able to express iterative parallel algorithms
- **User defined computation** consisting of:
  - **Update Functions**: local computation
  - **Sync Mechanism**: global aggregation

Outline

Graphlab Data Model

Consists of two parts:

1. **A directed data graph:**

$$G := (V, E)$$

where V is the set of vertices and $E \subseteq V \times V$ the directed edges

2. **A globally shared data table:**

$$\mathbf{T}[\text{Key}] \rightarrow \text{Value}$$

which maps keys to arbitrary blocks of data

Graphlab Data Model

Users can associate with each graph vertice and edge:

- Arbitrary **data blocks**
- Program or model **parameters**

Some notation:

Data associated with a vertex v: D_v

Data associated with an edge (u, v): $D_{u,v}$

Set of all *outbound* edges from v: $(v \rightarrow *)$

Set of all *inbound* edges from v: $(* \rightarrow v)$

Neighboring vertices of v: $\mathcal{N}_v := \{u | (u, v) \in E \vee (v, u) \in E\}$

GraphLab - User defined computation

There are two types of user defined computation on GraphLab:

- **Update function**:
  - local computation
  - executed on small neighborhoods
  - different functions may access and modify overlapping contexts
- **Sync Mechanism**:
  - global aggregation
  - runs concurrently with the update functions

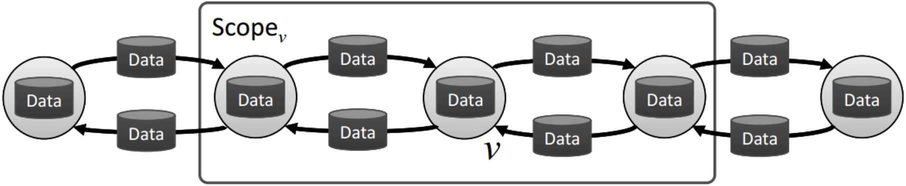
Outline

Update Function - Scope of a Vertex

The **Update Function** is a *stateless* user defined function

It operates on the **scope** $\mathcal{S}_v$ of a vertex v:

$$\mathcal{S}_v := \{v\} \cup (v \rightarrow *) \cup (* \rightarrow v) \cup \mathcal{N}_v$$



Once executed on a vertex v, it may trigger the execution of the update functions on other vertices $\mathcal{T}$

Update Function

Be $\mathcal{S}_v$ the scope of a vertex v and $\mathbf{T}$ the globally shared table.

The application of an update function f to a vertex v can be defined as

$$(D_{\mathcal{S}_v}, \mathcal{T}) \leftarrow f(D_{\mathcal{S}_v}, \mathbf{T})$$

Where:

- $D_{\mathcal{S}_v}$ is the data associated with the scope of v
- f has read-only access to $\mathbf{T}$
- $\mathcal{T}$ is a set of vertices

Update Function and Vertex Programs

In the most recent version of the GraphLab abstraction, Update Functions are called **Vertex Programs**

A Vertex Program implements the *Gather-Apply-Scatter (GAS)* model, and has three phases:

- **Gather:** executed in parallel on each edge of the current vertex (usually to read data)
- **Apply:** atomic function that modifies the vertex data
- **Scatter:** Executed in parallel on each edge of the current vertex (usually to update edge data and signaling neighboring nodes)

This decomposition allows us to execute a single vertex program on several machines simultaneously and move computation to the data.

Structure of an Update Function

procedure UPDATEFUNCTION
input: vertex v, scope $\mathcal{S}_v$, data $D_{\mathcal{S}_v}$, shared table $\mathbf{T}$

  GatherType: g ▷ stores the results of the gather phase
  for $(u, v) \in (* \to v)$ **do** ▷ In Parallel
    $g \leftarrow g + \text{gather}((u, v))$
  end for
  $\text{apply}(g, D_v)$
  for $(v, u) \in (v \to *)$ **do** ▷ In Parallel
    $\text{scatter}((v, u))$
  end for
end procedure

Outline

Sync Mechanism

Aggregates data accross all vertices in the graph

Associates the result with a particular entry in $\mathbf{T}$

The user provides:

- a key k
- an initial value r_k^0
- a fold function: $r_k^{i+1} \leftarrow \text{Fold}_k\left(D_v, r_k^i\right)$
- an optional merge function: $r_k^l \leftarrow \text{Merge}_k\left(r_k^i, r_k^j\right)$
- an apply function: $\mathbf{T}[k] \leftarrow \text{Apply}_k\left(r_k^{|V|}\right)$

Sync Mechanism

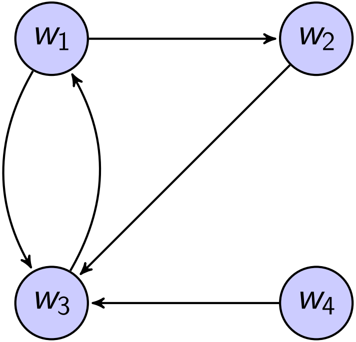
procedure SYNC ALGORITHM
input: key k, vertices V, data $\{D_v\}_{v \in V}$, initial value r_k^0, shared table $\mathbf{T}$

$t \leftarrow r_k^0$
for $v \in V$ **do**
$t \leftarrow \text{Fold}_k(D_v, t)$
end for
$\mathbf{T}[k] \leftarrow \text{Apply}_k(t)$
end procedure

Outline

Pagerank: Data Model

- $G := \{w_1, w_2, w_3, w_4\}$
- $E := \{(w_1, w_2), (w_1, w_3), (w_3, w_1), (w_2, w_3), (w_4, w_3)\}$
- $D_v := PR(v)$
- $D_{v,u} := \emptyset$



Pagerank: Update function

procedure PAGERANKUPDATEFUNCTION
input: vertex v, scope $\mathcal{S}_v$, ingoing edges $(* \to v)$

$\mathcal{T} \leftarrow \emptyset$
$D_v^{old} \leftarrow D_v$
$p \leftarrow 0$
for $u \in \{u | (u, v) \in (* \to v)\}$ **do**
  $p \leftarrow p + \frac{D_u}{|(u \to *)|}$
end for
$D_v \leftarrow (1 - \beta) + \beta * p$
if $|D_v - D_v^{old}| > \epsilon$ **then**
  $\mathcal{T} \leftarrow \mathcal{N}_v$
end if
return $(D_{\mathcal{S}_v}, \mathcal{T})$
end procedure

Pagerank: Vertex Program

procedure PAGERANKGATHER
input: vertex v, scope $\mathcal{S}_v$, ingoing edge $(u \to v)$

    return $\frac{D_u}{|(u \to *)|}$

end procedure

procedure PAGERANKAPPLY
input: vertex v, scope $\mathcal{S}_v$, gather result p,

    $D_v^{old} \leftarrow D_v$
    $D_v \leftarrow (1-\beta) + \beta * p$
    if $|D_v - D_v^{old}| > \epsilon$ **then**
        Signal to perform scatter
    end if

end procedure

Pagerank: Scatter

procedure PAGERANKSCATTER
input: outgoing edge $(v \rightarrow u)$, set of vertices to be rescheduled $\mathcal{T}$

Add u to $\mathcal{T}$

end procedure

PageRank Implementation: Data Types

```
struct web_page {
  std::string pagename;
  double pagerank;

  explicit web_page(std::string name):
          pagename(name),pagerank(0.0){ }

  void save(graphlab::oarchive& oarc) const{
     oarc << pagename << pagerank;
  }
  void load(graphlab::iarchive& iarc) {
     iarc >> pagename >> pagerank;
  }
};
typedef graphlab::distributed_graph
          <web_page, graphlab::empty> graph_type;
```

PageRank Implementation: Reading the Data

```
bool line_parser(graph_type& graph,
                 const std::string& filename, const std::string&
  std::stringstream strm(textline);
  graphlab::vertex_id_type vid;
  std::string pagename;

  strm >> vid >> pagename;

  graph.add_vertex(vid, web_page(pagename));

  while(1){
    graphlab::vertex_id_type other_vid;
    strm >> other_vid;
    if (strm.fail()) return true;
    graph.add_edge(vid, other_vid);
  }
  return true;
}
```

PageRank Implementation: Vertex Program

```
class pagerank_program :
  public graphlab::ivertex_program<graph_type, double>,
  public graphlab::IS_POD_TYPE {

  public:
  bool perform_scatter;

  edge_dir_type gather_edges(icontext_type& context,
                             const vertex_type& vertex);
  edge_dir_type scatter_edges(icontext_type& context,
                              const vertex_type& vertex);

  double gather(icontext_type& context,
                const vertex_type& vertex, edge_type& edge);
  void apply(icontext_type& context, vertex_type& vertex,
             const gather_type& total);

  void scatter(icontext_type& context,
               const vertex_type& vertex, edge_type& edge);
};
```

PageRank Implementation: Gather

```
// we are going to gather on all the in−edges
edge_dir_type gather_edges(icontext_type& context,
                           const vertex_type& vertex) const {

   return graphlab::IN_EDGES;

}

// for each in−edge gather the weighted sum of the edge.
double gather(icontext_type& context,
                     const vertex_type& vertex,
                     edge_type& edge) const {

   return    edge.source().data().pagerank
                  / edge.source().num_out_edges();

}
```

PageRank Implementation: Apply

```
// Use the total rank of adjacent pages
// to update this page
void apply(icontext_type& context,
           vertex_type& vertex,
           const gather_type& total) {

   double newval = total * 0.85 + 0.15;
   double oldval = vertex.data().pagerank;
   vertex.data().pagerank = newval;
   perform_scatter = (std::fabs(prevval - oldval) > 1E-3);

}
```

PageRank Implementation: Scatter

```
edge_dir_type scatter_edges(icontext_type& context,
                            const vertex_type& vertex) {

  if (perform_scatter) return graphlab::OUT_EDGES;
  else return graphlab::NO_EDGES;

}

void scatter(icontext_type& context,
             const vertex_type& vertex,
             edge_type& edge) const {

  context.signal(edge.target());

}
```

Outline

Execution Model

```
procedure GraphLabExec
input:  Graph G := (V, E), data D, initial vertex set 𝒯 ⊆ V

    while |𝒯| > 0 do
        v ← RemoveNext(𝒯)
        (𝒯′, D_{S_v}) ← f(D_{S_v}, T)
        𝒯 ← 𝒯 ∪ 𝒯′
    end while
end procedure
```

Serializability

- The GraphLab abstraction provides a rich *sequential model*
- Parallel Execution:
  - Multiple processors to execute the same loop on the same graph
  - Simutaneous execution of update functions on different vertices
- We desire a **serializable execution**:

Definition

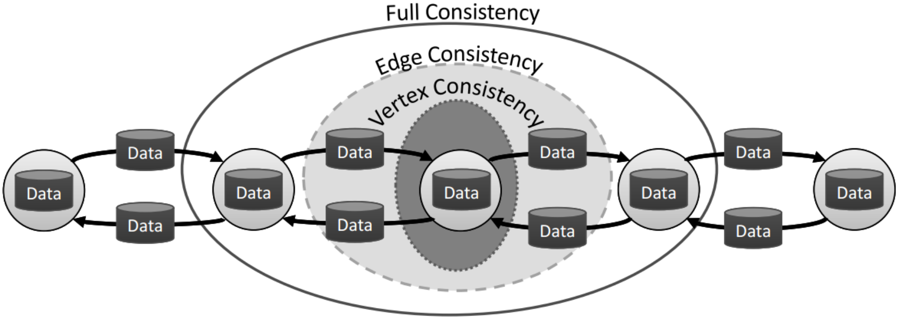
A GraphLab program is sequentially consistent if for every parallel execution, there exists a sequential execution of update functions that produces the same values in the data graph

Data Consistency

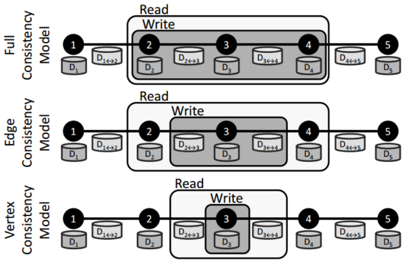
To ensure Serializability we need **consistency models**
GraphLab supports three types of consistency models:

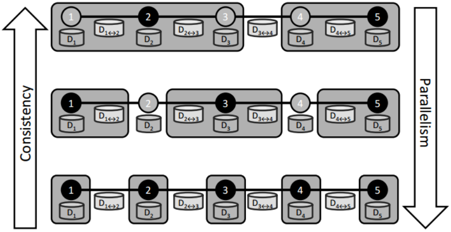
- **Full Consistency**: the scope of two concurrently executing update functions do not overlap
- **Edge Consistency**: each update function has R/W access to its vertex and adjacent edges but only read access to neighboring vertices
- **Vertex Consistency**: only ensures that no concurrently update functions are executed on the same vertice

Data Consistency - Exclusion Sets



Data Consistency



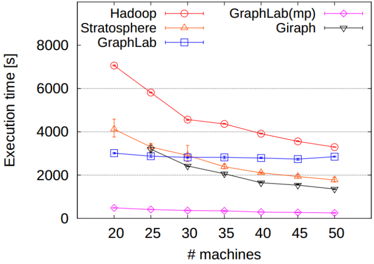


Data Consistency

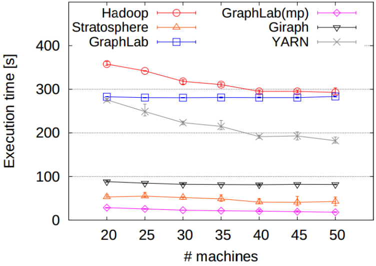
GraphLab *guarantees* sequential consistency under the following three conditions:

- The **full consistency** model is used
- The **edge consistency** model is used and update functions do not modify data in adjacent vertices
- The **vertex consistency** model is used and update functions only access local vertex data

Benchmarks - Horizontal Scaling



Dataset 1



Dataset 2

Dataset	$\|V\|$	$\|E\|$	Size
1	61.2K	50.9 M	655 MB
2	65.6 M	1.8 B	31 GB

Yong Guo, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, Theodore L. Willke. *Towards Benchmarking Graph-Processing Platforms*. Super Computing 2013

Summary

A GraphLab program consists of:

- A **data graph**
- An **update function**
  - Gather Function
  - Apply Function
  - Scatter Function
- A **sync mechanism**
- A **consistency model**