Distributed Graph Processing

OSCAR ROMERO

FACULTAT D'INFORMÀTICA DE BARCELONA

Centralised Graph Processing

Cost depends on the number of edges / nodes visited when processing it. Thus, it is affected by:

- The graph size and topology,
- The processing algorithm

But some times the graph is very large and the algorithm expensive

- E.g., Mining web 2.0, transportation routes, disease outbreak, bioinformatics, etc.

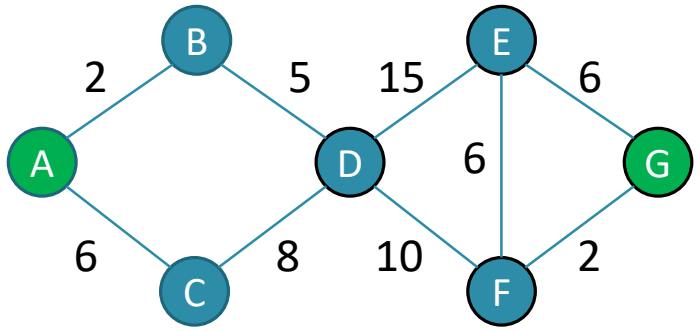
Navigational pattern matching, in the best case, it still has cubic computational complexity

Dijkstra's Shortest Path Algorithm

Invariant: *the subpath of any shortest path is itself a shortest path*
Triangle inequality: **$\delta(u,v) \leq \delta(u,x) + \delta(x,v)$**, if u,v is the shortest path

Greedy algorithm: *at each step, for each node x, choose the shortest available path from its neighbours*

Which node we were looking at when we updated the node's distance?



Source:
https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1206/lectures/dijkstra

Initial tbl	A	B	C	D	E	F	G
Distance to A	0	∞	∞	∞	∞	∞	∞
Previous	-						
Seen?	F	F	F	F	F	F	F

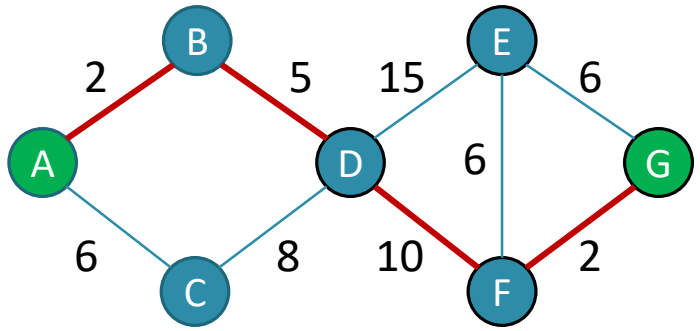
Step 1	A	B	C	D	E	F	G
Distance to A	0	2	6	∞	∞	∞	∞
Previous	-	A	A				
Seen?	T	F	F	F	F	F	F

Dijkstra's Shortest Path Algorithm

Invariant: *the subpath of any shortest path is itself a shortest path*
Triangle inequality: **$\delta(u,v) \leq \delta(u,x) + \delta(x,v)$**, if u,v is the shortest path

Greedy algorithm: *at each step, for each node x, choose the shortest available path from its neighbours*

Which node we were looking at when we updated the node's distance?



Source:
https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1206/lectures/dijkstra

Initial tbl	A	B	C	D	E	F	G
Distance to A	0	∞	∞	∞	∞	∞	∞
Previous	-						
Seen?	F	F	F	F	F	F	F

Final tbl	A	B	C	D	E	F	G
Distance to A	0	2	6	7	22	17	19
Previous	-	A	A	B	D	D	F
Seen?	T	T	T	T	T	T	T

To get the actual path, follow the *previous nodes* backwards from G

Graph Computation is Difficult to Parallelize

Graph computation is difficult to scale and parallelize

- **Data-driven computations.** The computations are driven by the vertex / edge. Structure of computations not known a-priori.
- **Unstructured problems.** Graphs are typically unstructured and highly irregular. Difficult to partition.
- **Poor locality.** The computations and data access patterns tend not to have very much locality. Yet, performance in contemporary processors is predicted upon exploiting locality.
- **High data access to computation ratio.** exploring the structure of a graph is more usual than performing large numbers of computations on the graph data.

Graph Computation is Difficult to Parallelize

Graph com

- **Data-d** … rtex / edge. Structu
- **Unstru** … nly irregular. Difficul
- **Poor lo** … to have very much l… licted upon exploit…
- **High da** … graph is more usual t… data.

In short, for sequential graph algorithms, which require random access to all the data, *poor locality* and the *indivisibility of the graph structure* cause time- and resource-intensive pointer chasing between storage mediums in order to access each datum.

In response to these shortcomings, new frameworks based on the *vertex-centric programming model* have been developed.

Graph Computation is Difficult to Parallelize

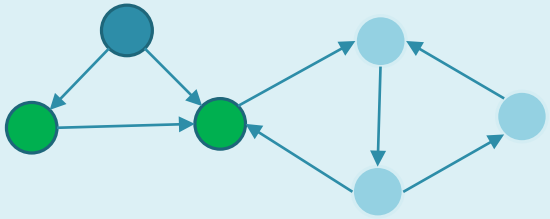
Graph com

- **Data-d**
  Structu
- **Unstru**
  Difficul
- **Poor lo**
  much l
  exploit
- **High da**
  usual t

In short, for sequential graph algorithms, which r
access to all the data, *poor locality* and the *indivis*
graph structure cause time- and resource-intensi
chasing between storage mediums in order to ac

In response to these shortcomings, new framewo
vertex-centric programming model have been de

As opposed to having a '**global**' perspective of the data (assuming all data is randomly accessible in memory), vertex-centric frameworks employ a **local**, vertex oriented perspective of computation, challenging the programmer to "**think like a vertex**" (TLAV)



Distributed Graphs

Remember to distinguish distributed **storage** and **distributed processing**

Storage

- Based on graph partitioning and using distributed storage such as HDFS, HBase or similar large-scale filesystems / databases
- Apache Titan as an off-the-shelf solution (on top of HBase or Cassandra)

Distributed processing requires graph data to be exposed in the form of (at least) two **views**:

- Set of vertices (or nodes)
- Set of edges (or links or relationships)

Distributed Processing

Thinking Like a Vertex (TLAV) frameworks

Based on Message Passing Interface (MPI) but adapted for graph processing

- It supports iterative execution of a user-defined vertex program over vertices of the graph
  - Vertices pass messages to adjacent vertices

They might either follow the Bulk Synchronous Parallel (BSP) computing model…

- Computation is based on supersteps
  - A superstep must finish before the next superstep starts (i.e., there is a synchronization barrier)

or an asynchronous computing model

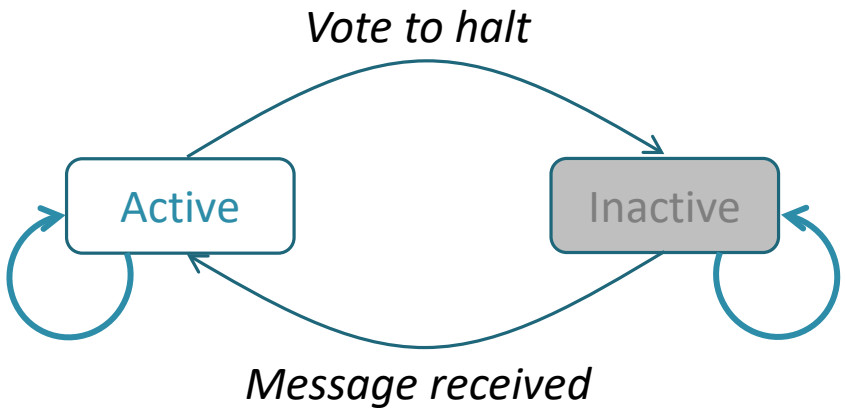
- Prone to suffer from deadlocks and / or data races / concurreny problems
- May improve performance under certain assumptions

Maximum Value Example

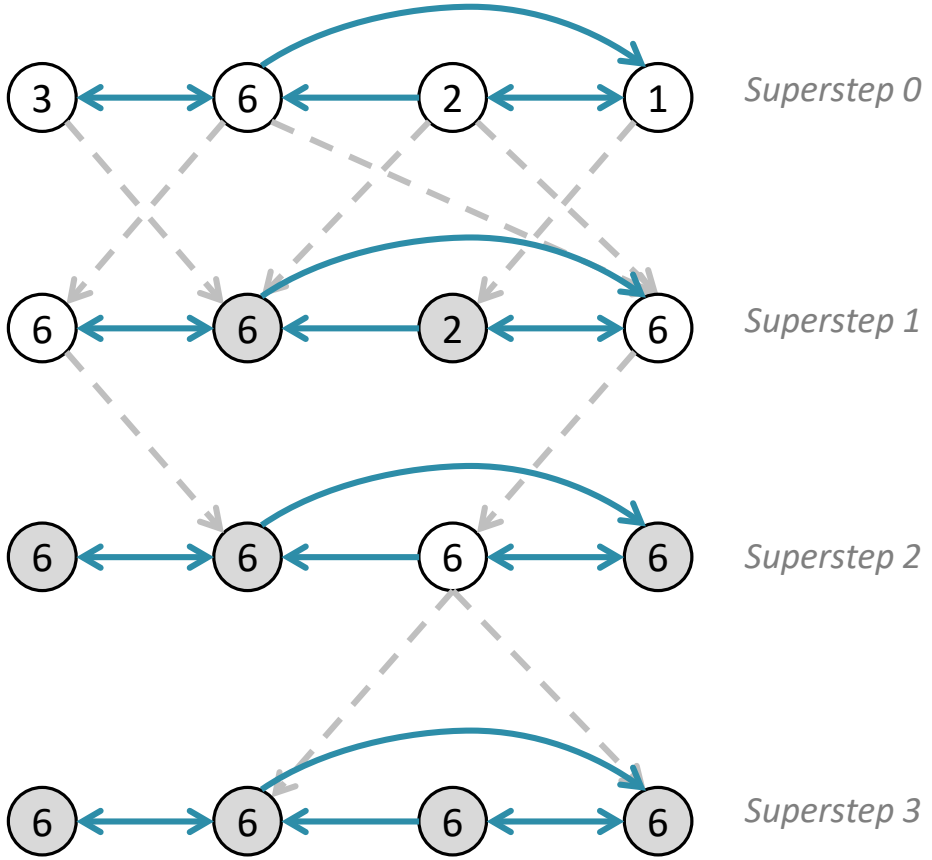
Calculate max using TLAV

In each superstep, any vertex that has learned a larger value from its messages sends it to all its neighbours.
When no further vertices change in a superstep, terminate.

Vertex state machine



Dotted lines are messages. Shaded vertices have voted to halt



Source: Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski: Pregel: a system for large-scale graph processing. SIGMOD Conference 2010: 135-146

BSP-Style Synchronization

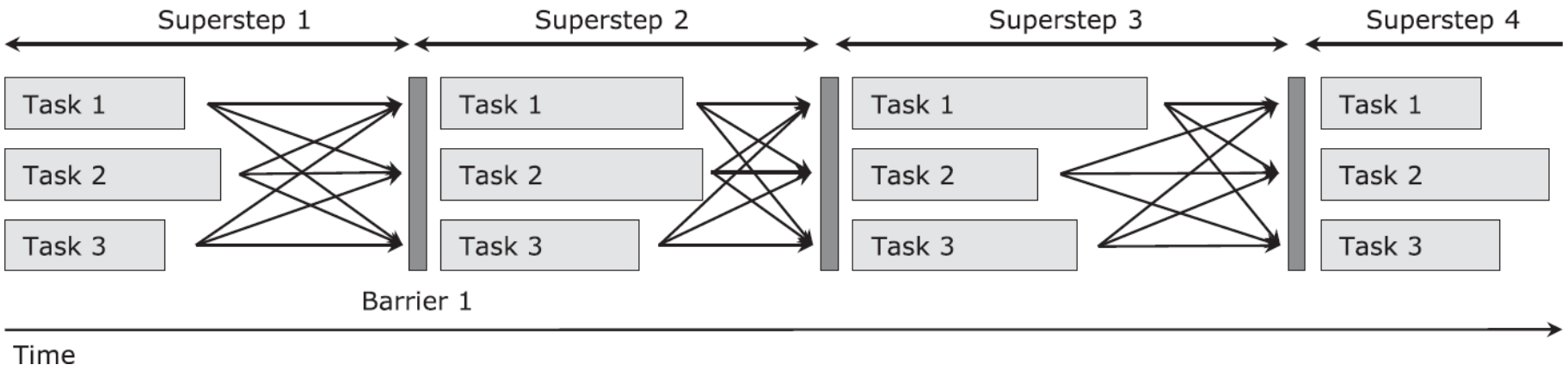
SYNCHRONIZED TLAV

Synchronized TLAV

A TLAV framework supports iterative execution of a user-defined vertex program over vertices of the graph

- Programs are composed of several interdependent components that drive program execution (**vertex kernels**)
- Components are only allowed to communicate via messages (MPI)
- A synchronization barrier is set between **interdependent** components
  - BSP *supersteps*

Synchronized TLAV (BSP-Style)



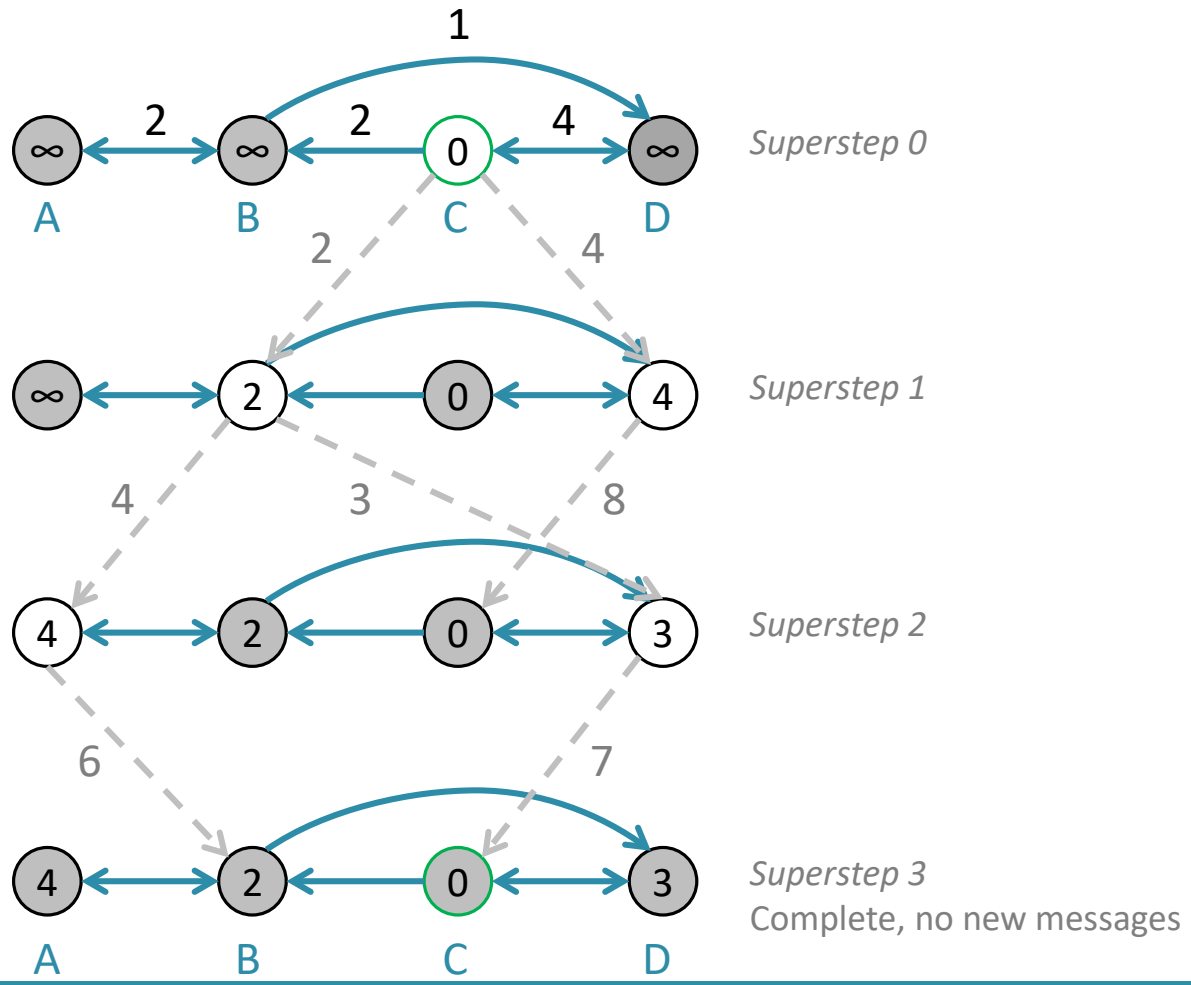
Example: Single-Source Shortest-Path (I)

ALGORITHM 1: Single-Source Shortest Path for a Synchronized TLAV Framework

input: A graph $(V, E) = G$ with vertices $v \in V$ and edges from $i \rightarrow j$ s.t. $e_{ij} \in E$, and starting point vertex $v_s \in V$

```
foreach v ∈ V do shrtest_path_len_v ← ∞;     /* initialize each vertex data to ∞ */
send (0, v_s);                               /* to activate, send msg of 0 to starting point */
repeat                                       /* The outer loop is synchronized with BSP-styled barriers */
    for v ∈ V do in parallel                 /* vertices execute in parallel */
        /* vertices inactive by default; activated when msg received */
        /* compute minimum value received from incoming neighbors */
        minIncomingData ← min(receive (path_length));
        /* set current vertex-data to minimum value */
        if minIncomingData < shrtest_path_len_v then
            shrtest_path_len_v ← minIncomingData;
            foreach e_vj ∈ E do
                /* send shortest path + edge weight to outgoing edges */
                path_length ← shrtest_path_len_v + weight_e;
                send (path_length, j);
            end
        end
        halt ();
    end
until no more messages are sent;
```

Example: Single-Source Shortest-Path (II)



Example: Single-Source TLAV Shortest-Path (I)

ALGORITHM 1: Single-Source Shortest Path for a Synchronized TLAV Framework

input: A graph $(V, E) = G$ with vertices $v \in V$ and edges from $i \to j$ s.t. $e_{ij} \in E$, and starting point vertex $v_s \in V$

foreach $v \in V$ **do** shrtest_path_len$_v \leftarrow \infty$;
send $(0, v_s)$;

Initializations

You just need to provide the **vertex kernel**:

```
1   minIncomingData ← min(receive (path_length));
    /* set current vertex-data to minimum value                    */
2   if minIncomingData < shrtest_path_len_v then
3       shrtest_path_len_v ← minIncomingData;
4       foreach e_vj ∈ E do
            /* send shortest path + edge weight to outgoing edges  */
5           path_length ← shrtest_path_len_v + weight_e;
6           send (path_length, j);
7       end
8   end
9   halt ();
```

Graph Distribution and Communications

SYNCHRONIZED TLAV

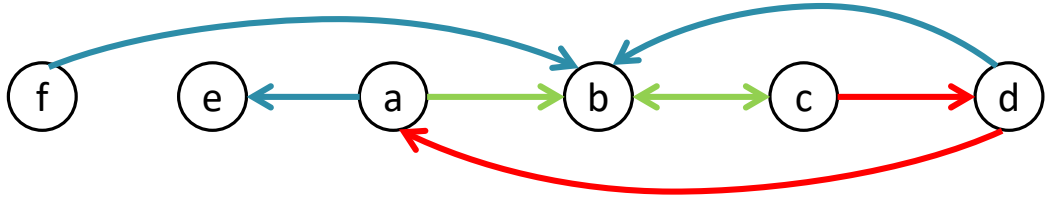
TLAV: Graph Distribution

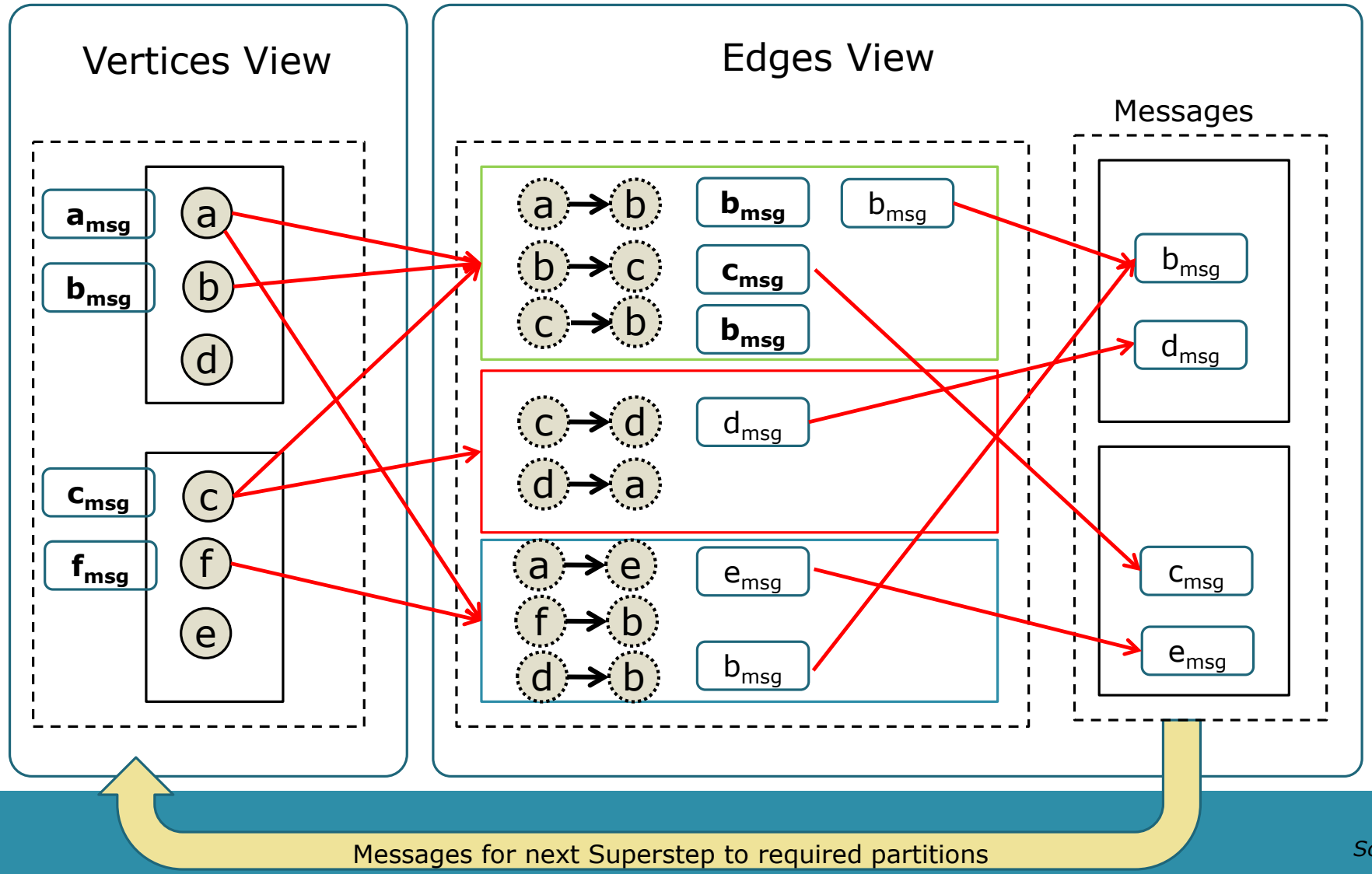
Recall TLAV graph processing requires graph data to be exposed in the form of two views:

- Set of vertices (or nodes)
- Set of edges (or links or relationships)

Let us consider now TLAV graph processing over distributed vertex / edge views

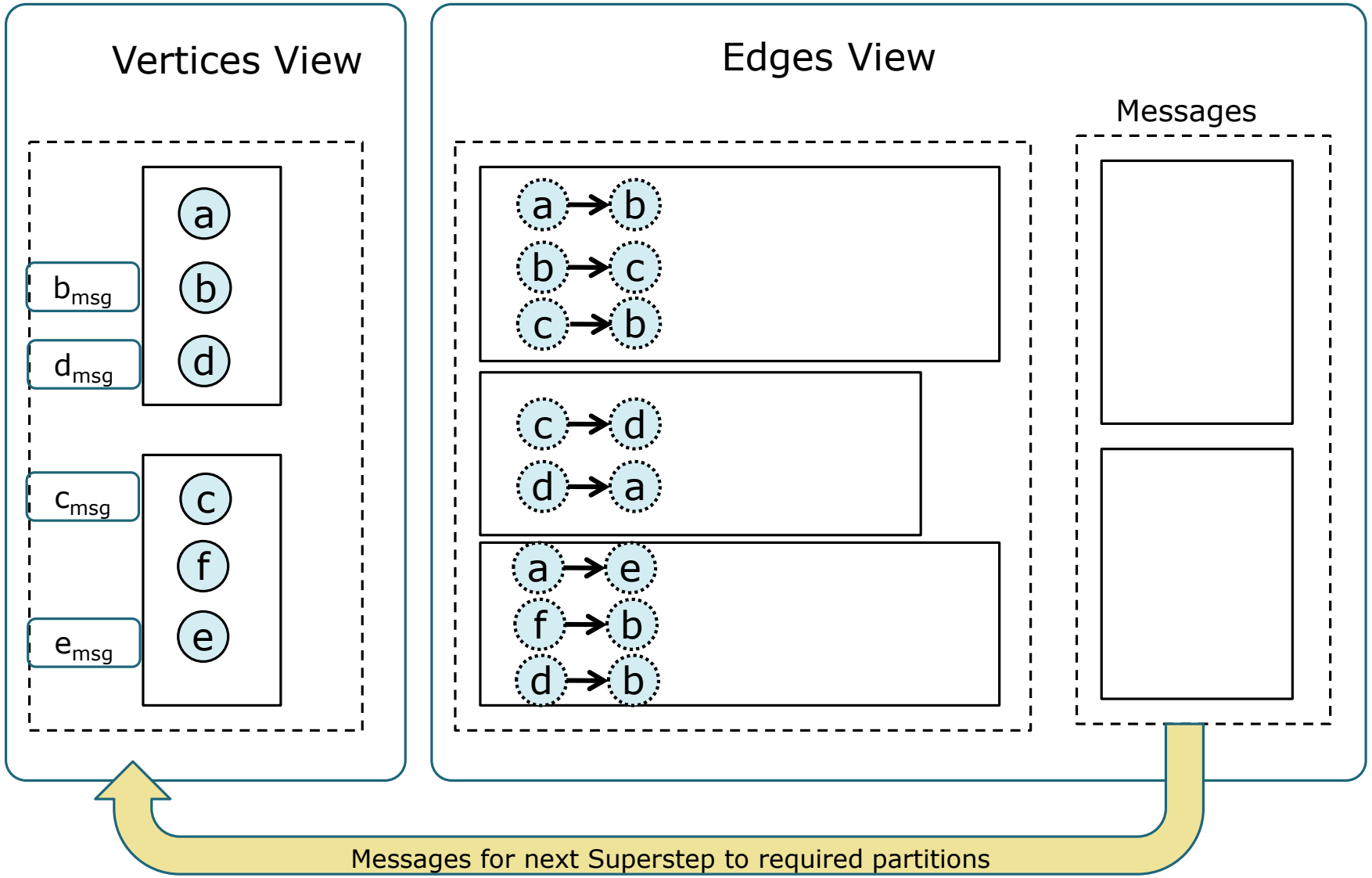
TLAV: Graph Distribution





Source: Rohit Kumar's PHD

Activity: Execute Next Superstep



Source: Rohit Kumar's PHD

Think Like a Vertex!

Think of independent vertices executing an arbitrary complex piece of code (vertex kernel)

- Edges are not first-class citizens as they have no computation associated (but they may have an associated state)
- **IMPORTANT:** states can **only** be shared through MPI

After the execution, vertices send messages to adjacent vertices

- Such messages can be arbitrary complex and may change the graph topology

The code must include a *halt condition* or *halt vote*

- Then, a node becomes inactive and does not pass messages anymore
- If a message is received, the node is automatically back to active

TLAV makes transparent to the user the graph distribution

Activity

Objective: Understand the TLAV approach

Modify the shortest path vertex kernel to include the information of the shortest path (and not only the number of hops)

- For example: from A to B the shortest path is 6 and the path is A-C-D-E-F-H-B

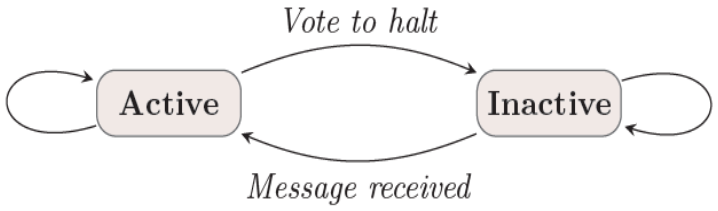
Pregel

Pregel is the most famous TLAV framework

- Developed by Google in 2010

Model of computation

- *Input*: directed graph (each vertex uniquely identified)
  - Three views required: list of nodes, directed edges and messages
  - Each node is identified uniquely by an id
- *Processing*: In each superstep the vertices compute in parallel
  - Either modify its state, that of its outgoing edges, receive messages (sent to it in the previous superstep), send messages to other vertices (to be received in the next superstep) or even mutate the topology of the graph
  - The algorithm finishes based on a *voting to halt*



- *Output:* Set of values explicitly output by the vertices

GraphX

It is a subproject within Apache Spark

- Built as a Spark module
- It follows Pregel's principles
- It follows the same idea as Spark GraphFrames to provide Pregel required views
  - Vertices, edges and triplet views (an edge + **its** nodes info; i.e., is a view of the whole edge)
- It provides a library with typical distributed algorithms
  - For example, PageRank, Connected components, Label propagation, SVD++, Strongly connected components, Triangle count, etc.

CONCLUSIONS

Comparison with MapReduce / Spark

There are some frameworks extending MapReduce to support iterative execution. Spark is naturally suited for iterating

However, they are not originally thought for iterating over graphs

- The topological graph information, even if static, **must be transferred** from mappers to reducers (or among Spark operations)
- This yields a significant network overhead

As result, TLAV outperforms MR-like programming models as it is specifically devised for graphs

Summary

TLAV allows to perform large-scale graph processing and have been massively used in practice

- The most prominent solution is MPI-based BSP-TLAV solutions

Is BSP-TLAV computationally complete? Note ***shared-memory reads*** are not allowed in this model! (intuition says yes, it is, not yet proved)

References

- Robert Ryan McCune, Tim Weninger, Greg Madey: **Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing**. *ACM Comput. Surv.* 48(2): 25:1-25:39 (2015)
- Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski: **Pregel: a system for large-scale graph processing**. *SIGMOD Conference 2010*: 135-146 (2010)