DISTRIBUTED GRAPH PROCESSING

Centralised Graph Processing

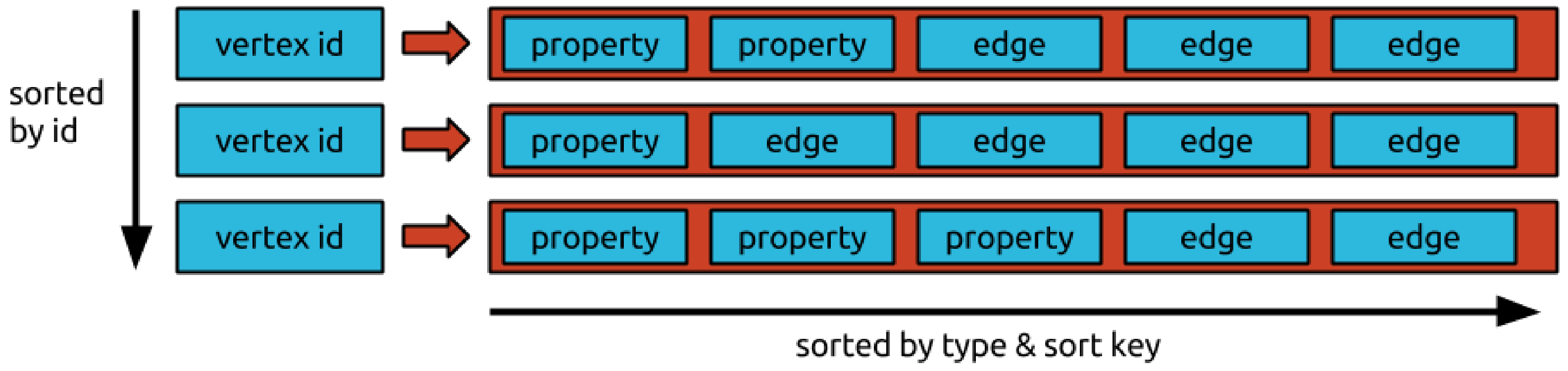
- Cost depends on the number of edges / nodes visited when processing it. Thus, it is affected by:
  - The graph size and topology,
  - The processing algorithm and the input pattern
- But some times the graph is very large and the algorithm expensive
  - E.g., Mining web 2.0, transportation routes, disease outbreak, bioinformatics, etc.
- Recall that navigational pattern matching, in the best case, it still suffers from cubic computational complexity

Distributed Graphs

- We must distinguish distributed storage and distributed processing
- Storage
  - Using HDFS, HBase or similar large-scale filesystems / databases
  - Apache Titan as an off-the-shelf solution
- 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)
  - Same principles as per distributed data management applies for each of these views
    - Partitioning, replicas, etc.

Distributed Storage

- Apache Titan: (a key-value-based incidence list)



- Works on top of Cassandra or HBase
- Properties and edges are stored as column:value
  - Sort key ~ key design

Distributed Processing

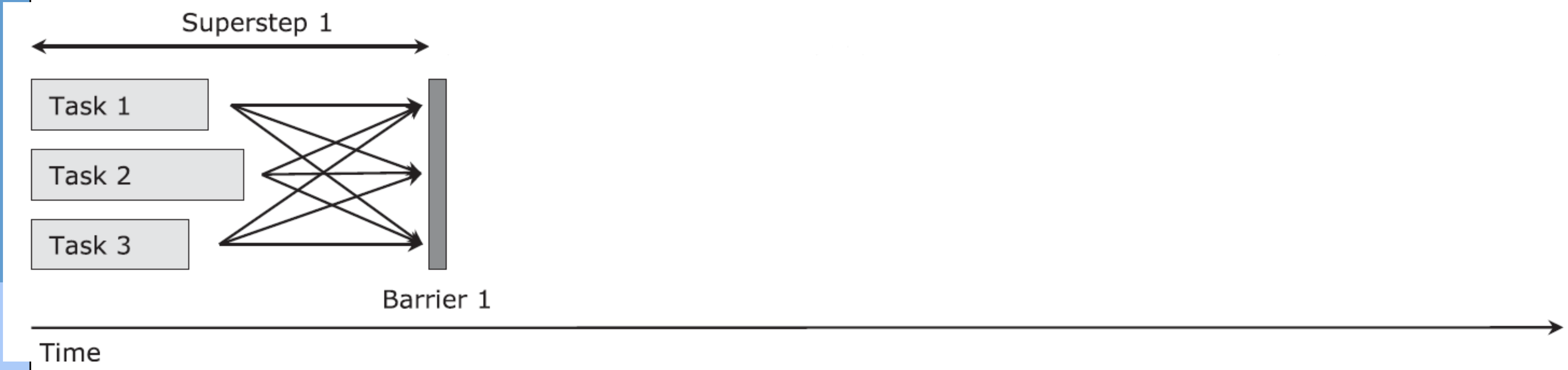
- Thinking Like a Vertex (TLAV) frameworks
  - Based on Message Passing Interface but adapted for graph processing
    - It supports iterative execution of a user-defined vertex program over vertices of the graph
      - Vertices pass messages to adjacent vertices
  - They might either follow the Bulk 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 ansynchronous computing model
    - Prone to suffer from deadlocks and / or data races / concurreny problems
    - May improve performance under certain assumptions

BSP-Style Synchronization

SYNCHRONIZED TLAV

Oscar Romero

Synchronized TLAV (BSP-Style)



Synchronized TLAV

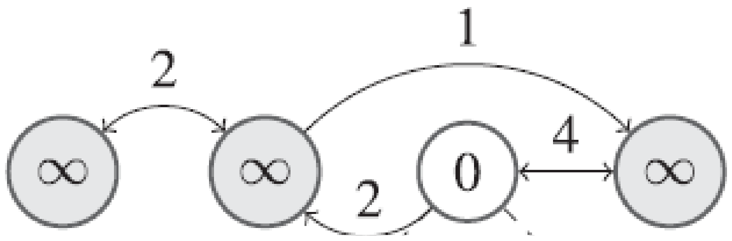
- A TLAV framework supports iterative execution of a user-defined vertex program over vertices of the graph
  - Programs are composed of several interdependent components that drive program execution (**vertex kernels**)
  - A synchronization barrier is set between interdependent components
    - BSP *supersteps*

Example: 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, $v_s$);
repeat
    for $v \in V$ do in parallel
        minIncomingData ← min(receive (path_length));
        if minIncomingData < shrtest_path_len$_v$ then
            shrtest_path_len$_v$ ← minIncomingData;
            foreach $e_{vj} \in E$ do
                path_length ← shrtest_path_len$_v$+weight$_e$;
                send (path_length, $j$);
            end
        end
        halt ();
    end
until no more messages are sent;
```

Example: Shortest-Path (II)

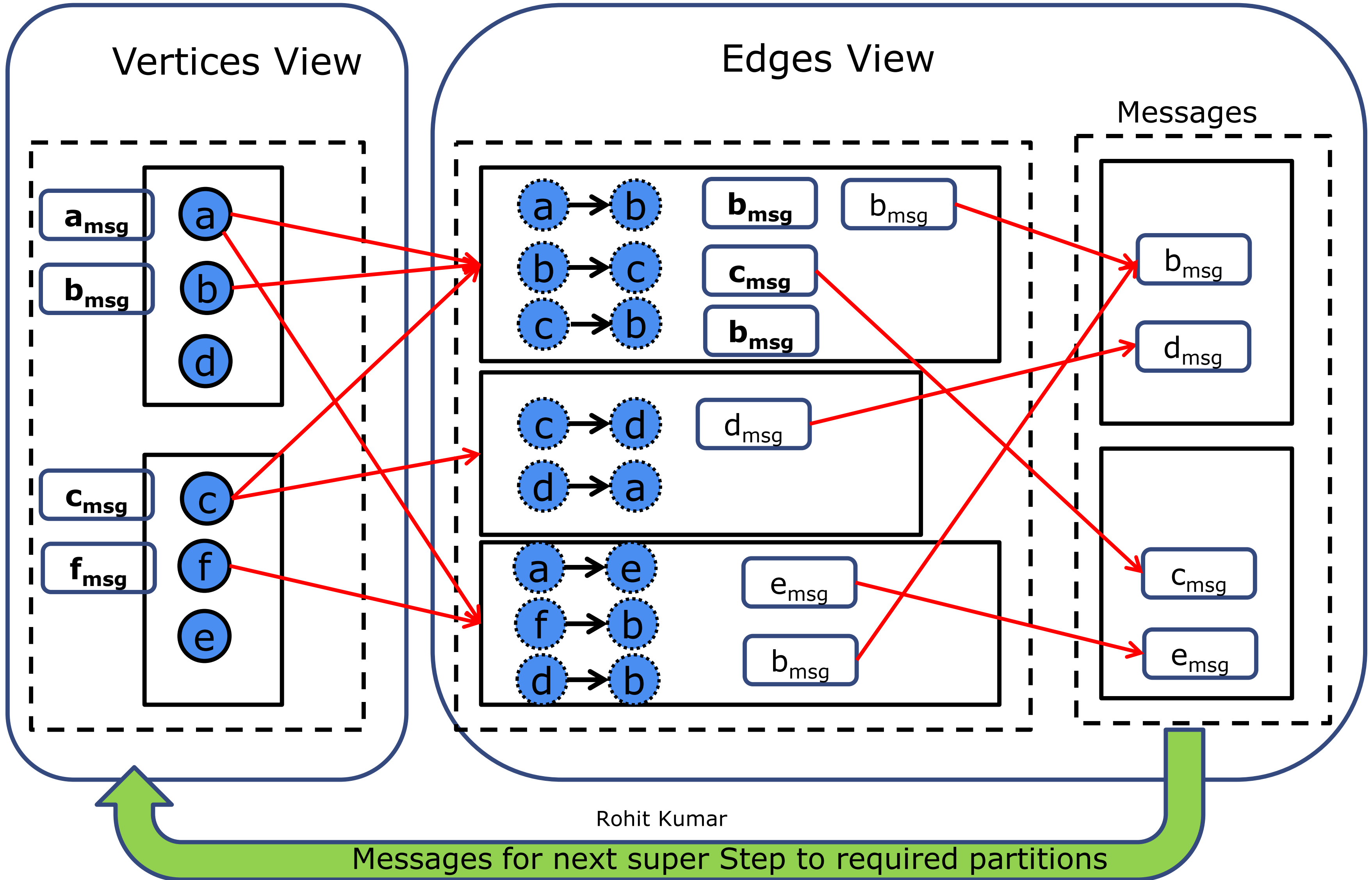


Example: TLAV Shortest-Path (I)

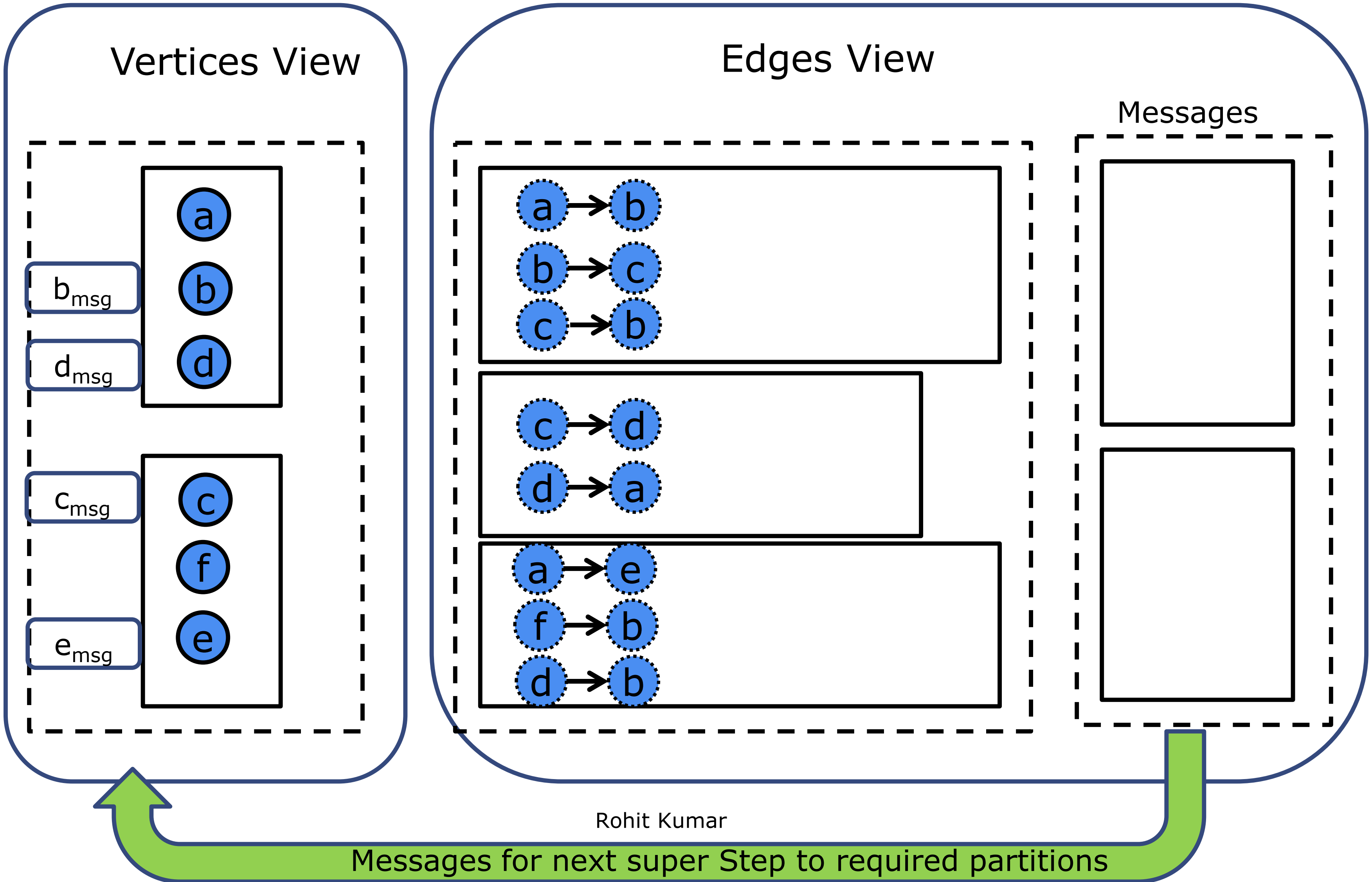
You just need to provide the vertex kernel:

```
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 $e_{vj} \in E$ do
        /* send shortest path + edge weight to outgoing edges */
        path_length ← $shrtest\_path\_len_v + weight_e$;
        send (path_length, $j$);
    end
end
halt ();
```

TLAV: Graph Distribution



Activity: Execute Next Superstep

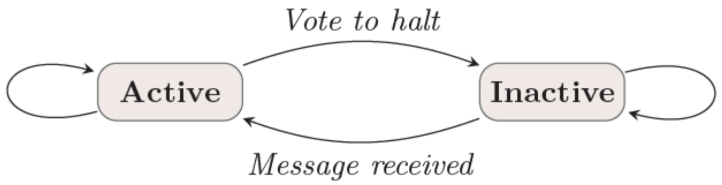


Think Like a Vertex!

- 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)
  - **<u>IMPORTANT</u>:** states can **only** be shared through MPI
- After the execution, vertices send messages to adjacent vertices
  - Such messages can be arbitrary complex and, for example, may change the graph topology
- The code must include a *halt condition* or *halt vote*
  - Then, a node becomes inactive and do 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

Pregel

- Pregel is the most famous BDS 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
    - Although it only allows to send messages to adjacent vertices
  - 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.

Scheduling, partitioning, communication and execution model

SCRUTINIZING TLAV

These slides are based on: *McCune et al. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing*

Oscar Romero

Pillars of TLAV Frameworks

- ***Scheduling***: How user-defined vertex programmes are scheduled for execution
- ***Communication***: How vertex program data is made accesible to other vertex programs
- ***Execution Model***: Implementation of vertex program execution and flow of data
  - We will follow a vertex-centric model but edge-centric is also possible
- ***Partitioning***: How the stored graph is stored across memory of the system's multiple working machines

Frameworks: Scheduling

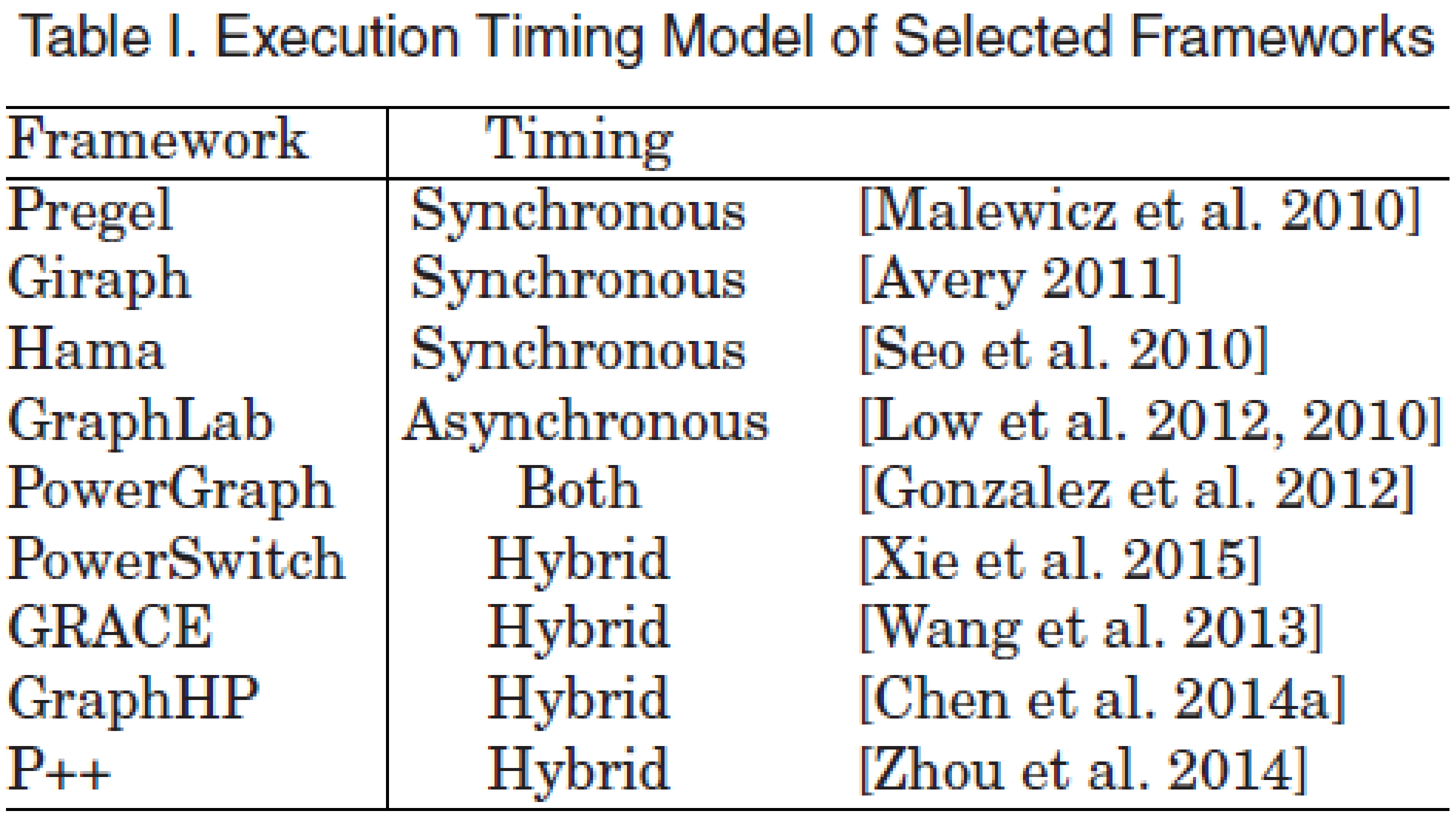


Table I. Execution Timing Model of Selected Frameworks

Framework	Timing	
Pregel	Synchronous	[Malewicz et al. 2010]
Giraph	Synchronous	[Avery 2011]
Hama	Synchronous	[Seo et al. 2010]
GraphLab	Asynchronous	[Low et al. 2012, 2010]
PowerGraph	Both	[Gonzalez et al. 2012]
PowerSwitch	Hybrid	[Xie et al. 2015]
GRACE	Hybrid	[Wang et al. 2013]
GraphHP	Hybrid	[Chen et al. 2014a]
P++	Hybrid	[Zhou et al. 2014]

Synchronous Scheduling: Pros

- The *synchronous* timing model is based on the original BSP processing model
  - In this model, active vertices are executed conceptually in parallel over one or more iterations, called *supersteps*
    - Synchronization is achieved through a global synchronization *barrier* situated between each superstep that blocks vertices from computing the next superstep until all workers complete the current superstep
    - Each worker coordinates with the master to progress to the next superstep
  - Synchronization is achieved because the barrier ensures that each vertex within a superstep has access to only the data from the previous superstep
    - Within a single processing unit, vertices can be scheduled in a fixed or random order because the execution order does not affect the state of the program
  - ***Pros:*** Synchronous systems are conceptually simple and perform exceptionally well for certain classes of algorithms
    - Almost always deterministic, making synchronous applications easy to design, program, test, debug, and deploy
  - ***Pros***: Demonstrate scalability
    - Often linear in the number of vertices
    - Can benefit from batch messaging between supersteps (merging)

Synchronous Scheduling: Cons

- ***Cons:*** the system throughput must remain high in each superstep to justify the cost of synchronization
  - For example, synchronization in the shortest-path algorithm for a highly partitioned graph accounts for over 80% of the total running time
  - Throughput is affected by:
    - The number of active vertices drops or the workload among workers becomes imbalanced, system resources can become underutilized
    - Iterative algorithms often suffer from “the curse of the last reducer” (aka *straggler* problem), where many computations finish quickly but a small fraction of computations take a disproportionately longer amount of time
    - Each superstep takes as long as the slowest vertex, so synchronous systems generally favor lightweight computations with small variability in runtime
- ***Cons:*** synchronous algorithms may not converge for some graph topologies
  - In general, algorithms that require some type of neighbour coordination may not always converge with the synchronous scheduling model without the use of some extra logic in the vertex program
  - In graph colouring algorithms, for example, vertices attempt to choose colors different from adjacent neighbors and require coordination between neighboring vertices. However, during synchronous execution, two neighboring vertices continually flip between each other’s color

Asynchronous Scheduling

- No explicit synchronization points (i.e., barriers) are provided, so any active vertex is eligible for computation whenever processor and network resources are available
- Vertex execution order can be dynamically generated and reorganized by the scheduler, and the "*straggler*" problem is eliminated
- As a result, many asynchronous models outperform corresponding synchronous models, but at the expense of added complexity

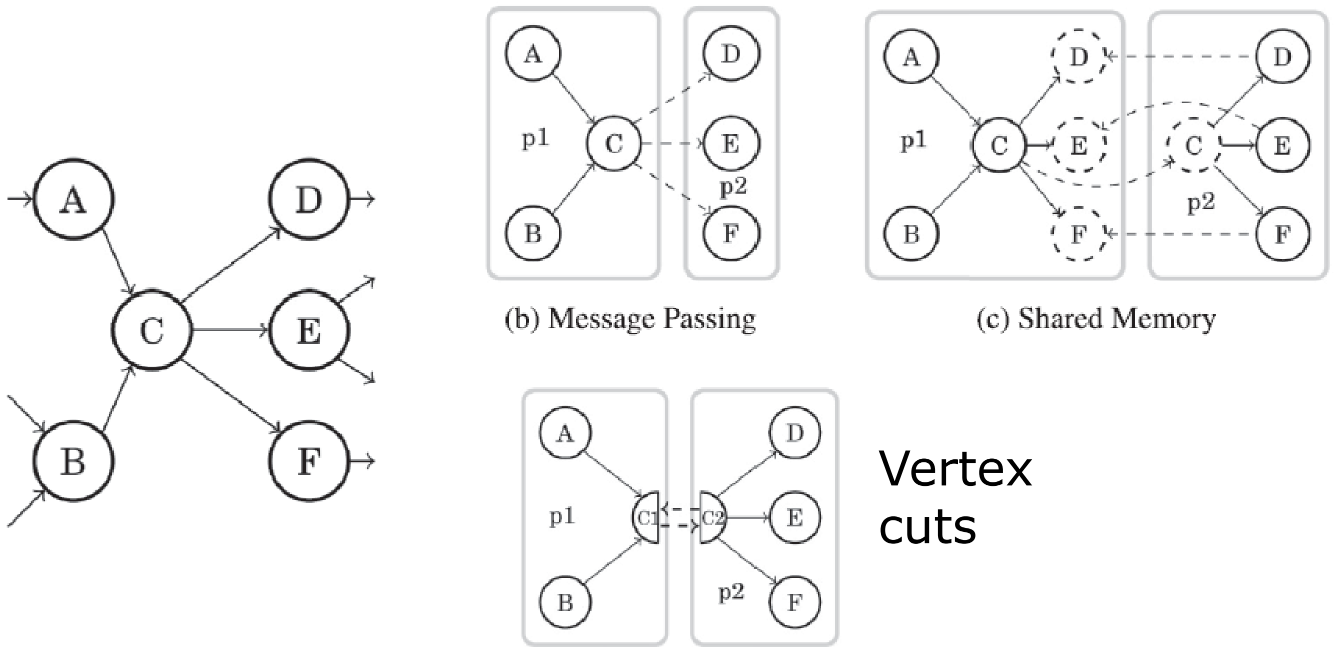
Asynchronous Scheduling

- ***Pros***: asynchronous systems especially outperform synchronous systems when the workload is imbalanced
  - For example, when computation per vertex varies widely, synchronous systems must wait for the slowest computation to complete, while asynchronous systems can continue execution maintaining high throughput
  - Many well-known algorithms exhibit asymmetric convergence
    - For PageRank, most nodes converge in one superstep while few of them (<3%) need more than 10 supersteps

- ***Cons***: It cannot take advantage of batch messaging optimizations
- ***Cons***: The typical pull model execution may result in unnecessary processing
- ***Cons***: Asynchronous algorithms face more difficult scheduling problems and also consistency issues (due to race data problems)

- In general, **synchronous execution generally accommodates I/O-bound algorithms**, while **asynchronous execution well-serves CPU-bound algorithms** by adapting to large and variable workloads

Communication



(b) Message Passing

(c) Shared Memory

Ghost nodes

Vertex cuts

(a) Sample Graph

(d) Shared Mem w/Vertex-Cuts

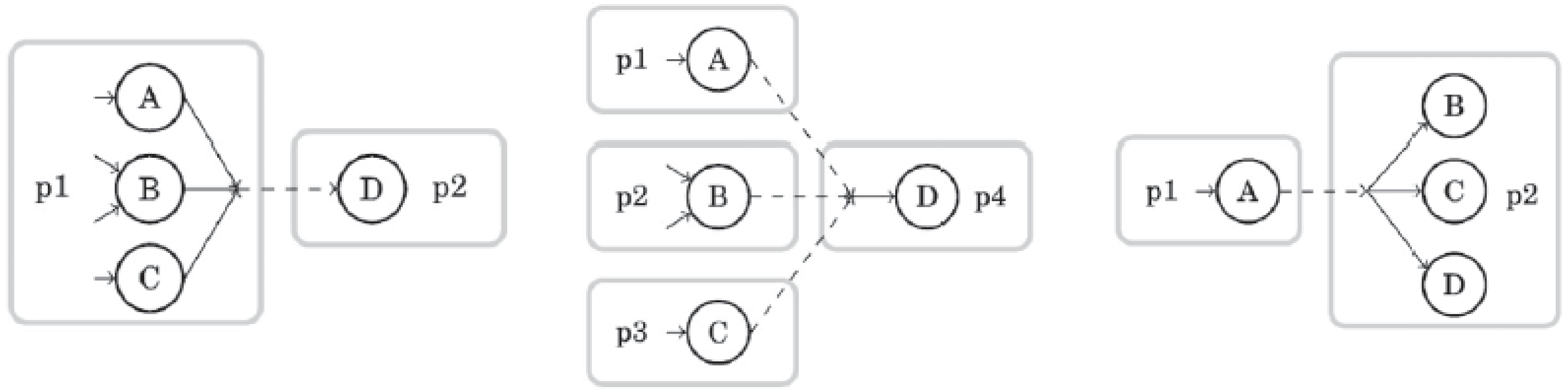
Fig. 4. Distributed communication patterns for common communication implementations. The sample graph is partitioned across two machines (see Section 3.4), with vertices A, B, and C residing on machine p1, and vertices D, E, and F on machine p2. Pregel is represented in (b), GraphLab in (c), PowerGraph in (d)

Communication: Message Passing

- Information is sent from one vertex program kernel to another via a message
- A message contains local vertex data and is addressed to the ID of the recipient vertex
  - A message can be addressed anywhere, but because vertices do not have ID information of all of the other vertices, destination vertex IDs are typically obtained by iterating over outgoing edges
- After computation is complete and a destination ID for each message is determined, the vertex dispatches messages to the local worker process
  - The worker process determines whether the recipient resides on the local machine or a remote one
    - In the case of the former, the worker process can place the message directly into the vertex's incoming message queue
    - Otherwise, the worker process looks up the worker ID of the destination vertex and places the message in an outgoing message buffer
      - Message buffers are flushed when they reach a certain capacity, sending messages over the network in batches. In principle, it tries to wait until the end of a superstep to send all messages in batch-mode

Message Passing Optimizations

- 3 main strategies:



(a) Sender-side Combiner (b) Receiver-side Combiner (c) Receiver-side Scatter

Communication: Shared-Memory

- Shared memory exposes vertex data as shared variables that can be directly read or modified by other vertex programs
  - Shared memory avoids the additional memory overhead constituted by messages
- This is typical of centralized graph processing, but some distributed systems apply it
- **Main problem**: For shared-memory TLAV frameworks, race conditions may arise when an adjacent vertex resides on a remote machine. Shared-memory TLAV frameworks often ensure memory consistency through mutual exclusion by requiring serializable schedules
  - Serializability, in this case, means that every parallel execution has a corresponding sequential execution that maintains consistency, *cf.*, the dining philosophers problem
- Most prominent solutions:
  - In GraphLab, border vertices are provided locally cached *ghost* copies of remote neighbors, where consistency between ghosts and the original vertex is maintained using pipelined distributed locking
  - In PowerGraph and GiraphX, graphs are partitioned by edges and cut along vertices, where consistency (i.e., serialization) across cached *mirrors* of the cut vertex is maintained using parallel Chandy-Misra locking
- The reduced overhead of shared memory compared to message passing may lead to 35% faster converges when computing PageRank on a large Web Graph

Partitioning

- Large-scale graphs must be divided into parts to be placed in distributed memory. Good partitions often lead to improved performance, but expensive strategies can end up dominating processing time, leading many implementations to incorporate simple strategies, such as random placement
- Effective partitioning evenly distributed the vertices for balanced workload while minimizing interpartition edges to avoid costly network traffic, a problem formally known as *k-way graph partitioning* that is NP-complete with no fixed-factor approximation
  - Leading work in graph partitioning can be broadly characterized as (1) rigorous but impractical mathematical strategies or (2) pragmatic heuristics used in practice

CURRENT OPEN PROBLEM

CONCLUSIONS

Comparison with MR / Spark

- There are some frameworks extending MapReduce to support iterative execution. Spark is naturally suited for iterating
  - MapReduce is based on Functional Programming
- 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, but 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