High Performance Graph Engine: New Application and Architecture Opportunities

System Research Group, MSR-Asia
In collaboration with MSR-SVC

Motivation

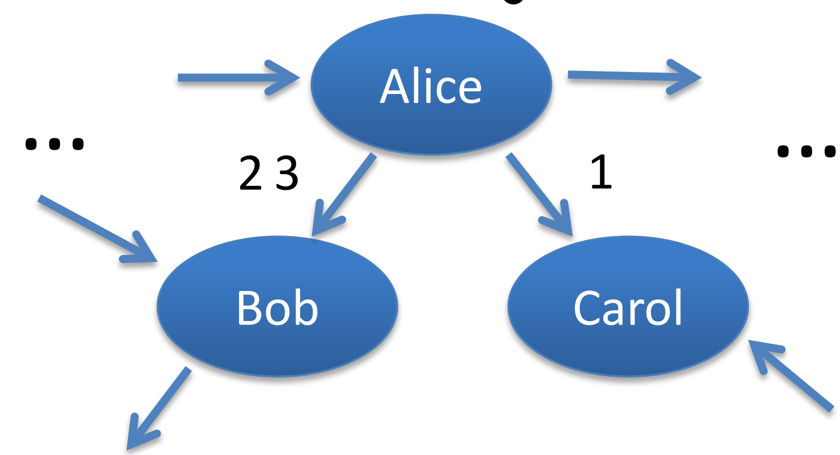
- Tremendous increase in graph data and apps
  - Graph mining on web graph and social network
  - Real-time graph query, e.g., knowledge graph
- Opportunities for research on graph engine
  - New scenario: analysis on a fast changing graph
  - Multicore server: graph-aware optimization

- Kineograph: taking the pulse of a fast-changing and connected world (Eurosys’12)
- Grace: managing large graphs on multi-cores with graph awareness (USENIX ATC’12)

Kineograph Background

- The age of real-time data –
  - New time-sensitive data generated continuously
  - Rich connections between entities
- Example: mention graph

Alice: You should see this @bob @carol

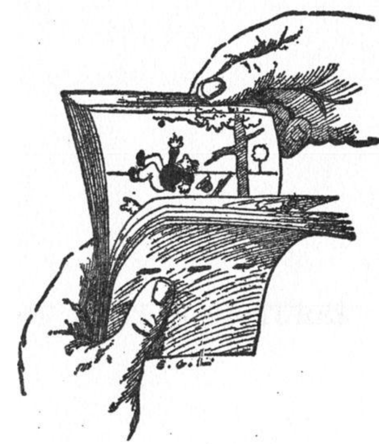


Goal: Compute Global Properties on the Changing Graph

System Challenges

- **High rate** of graph updates
- **Consistent** graph data
- **Static** graph algorithm vs. **changing** graph
- **Timely** results reflecting graph updates
- **Fault tolerant**

Kineograph: In-Memory Graph Store



THE KINEOGRAPH.

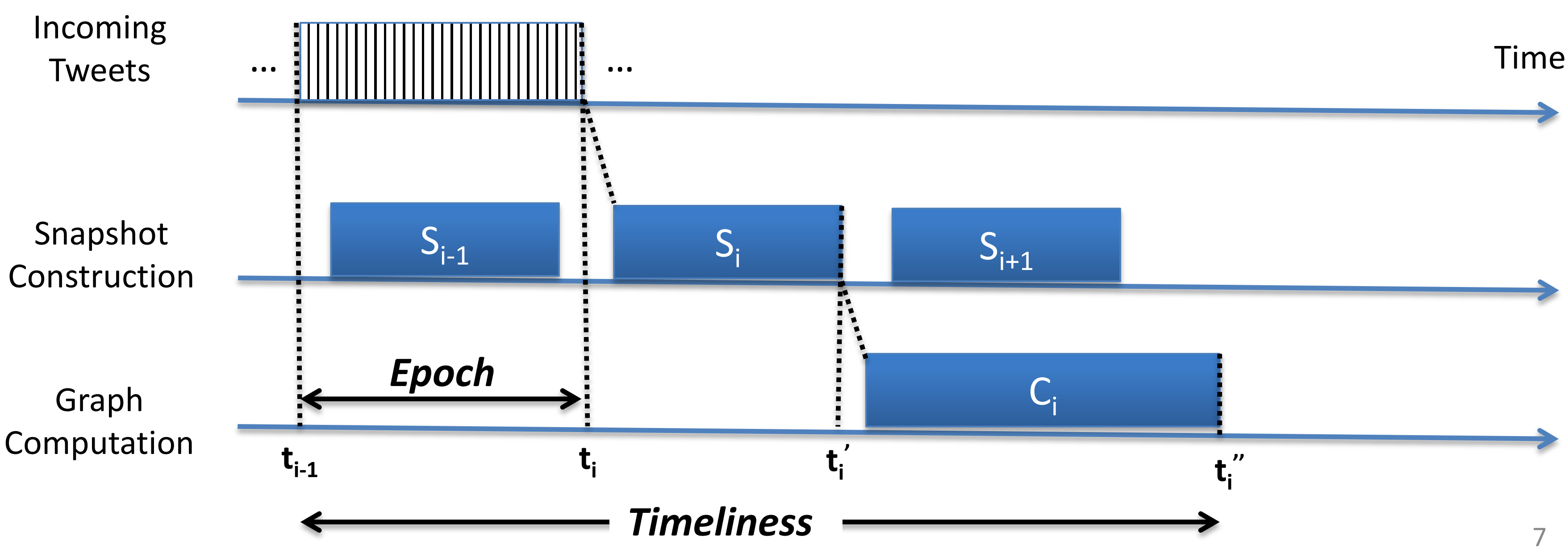
Zeitgenossische Illustration (1886)

Scalable and fault-tolerant distributed system for nearline graph mining

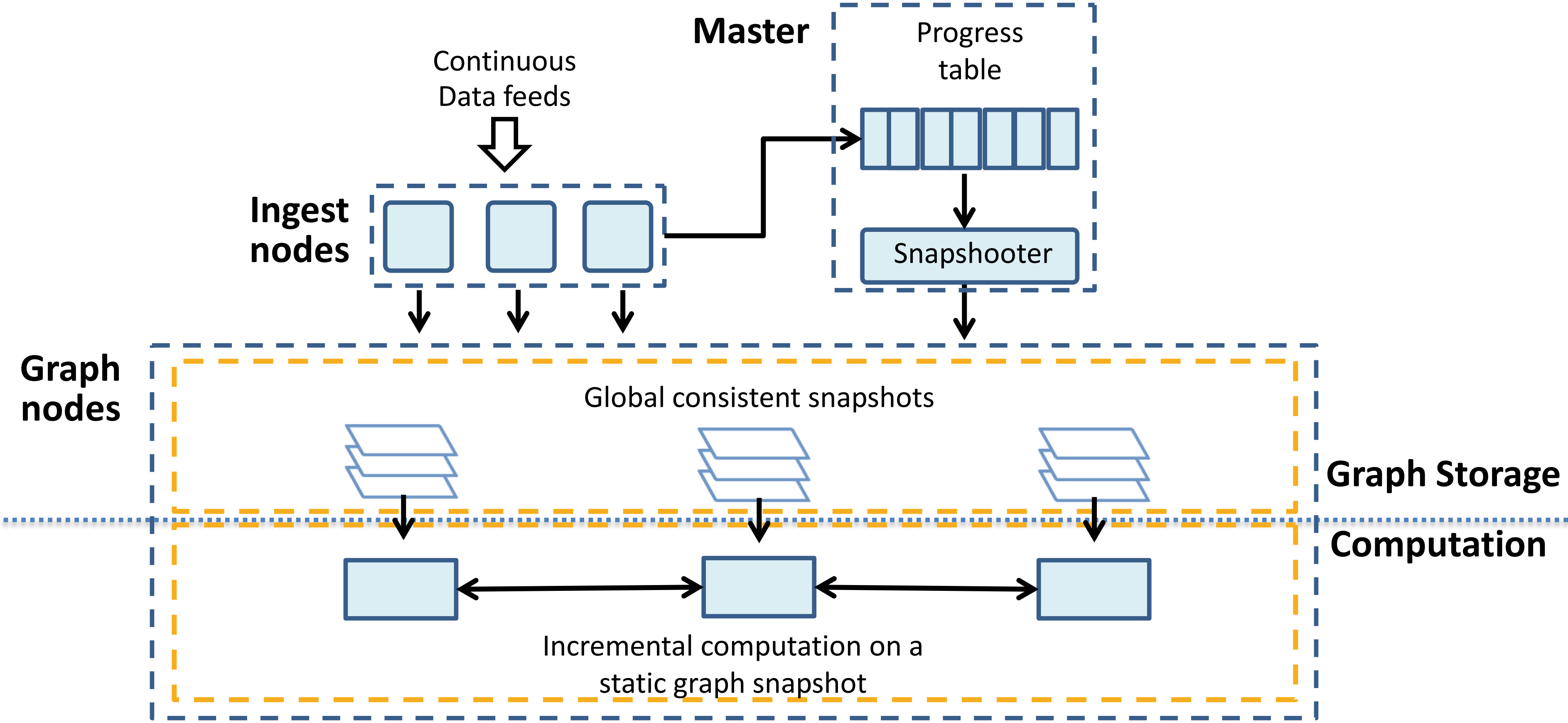
- Built-in support for incremental computation
  - Kineograph API for various graph algorithms
  - Examples:
    - InfluenceRank
    - Approximate all-pairs shortest paths
    - K-exposure (hash-tag histogram)
- Epoch commit protocol
  - Fast graph update and consistent snapshot production
  - Static graph algorithm operating over a snapshot

Graph Update/Compute Pipeline

- Multiple parallel data sources
- Graph update in **transaction**
  - E.g., tweet → updates of multiple edges/vertices, cross-partition operations
- Produce a consistent global snapshot periodically



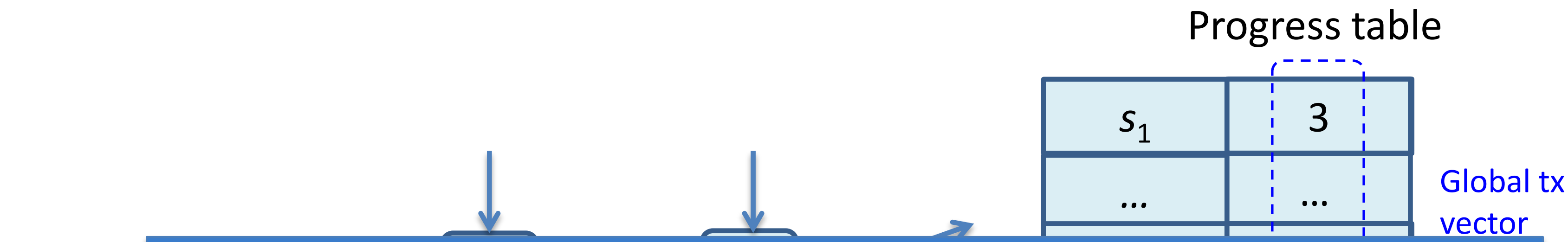
System Architecture



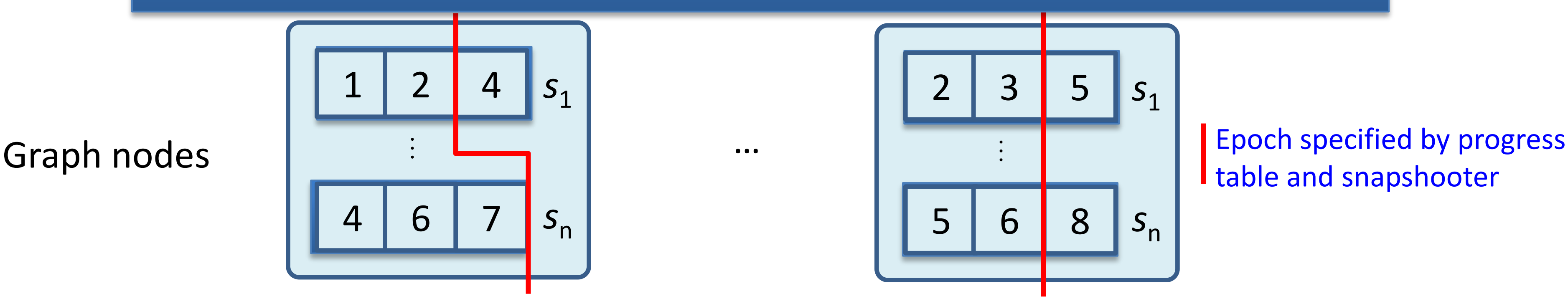
Key decision: **separation** of graph construction from graph computation

- Give rise to the epoch commit protocol
- Enable simple and separate fault tolerant mechanisms for graph update (quorum-based replication) and graph computation (check-point and primary backup)

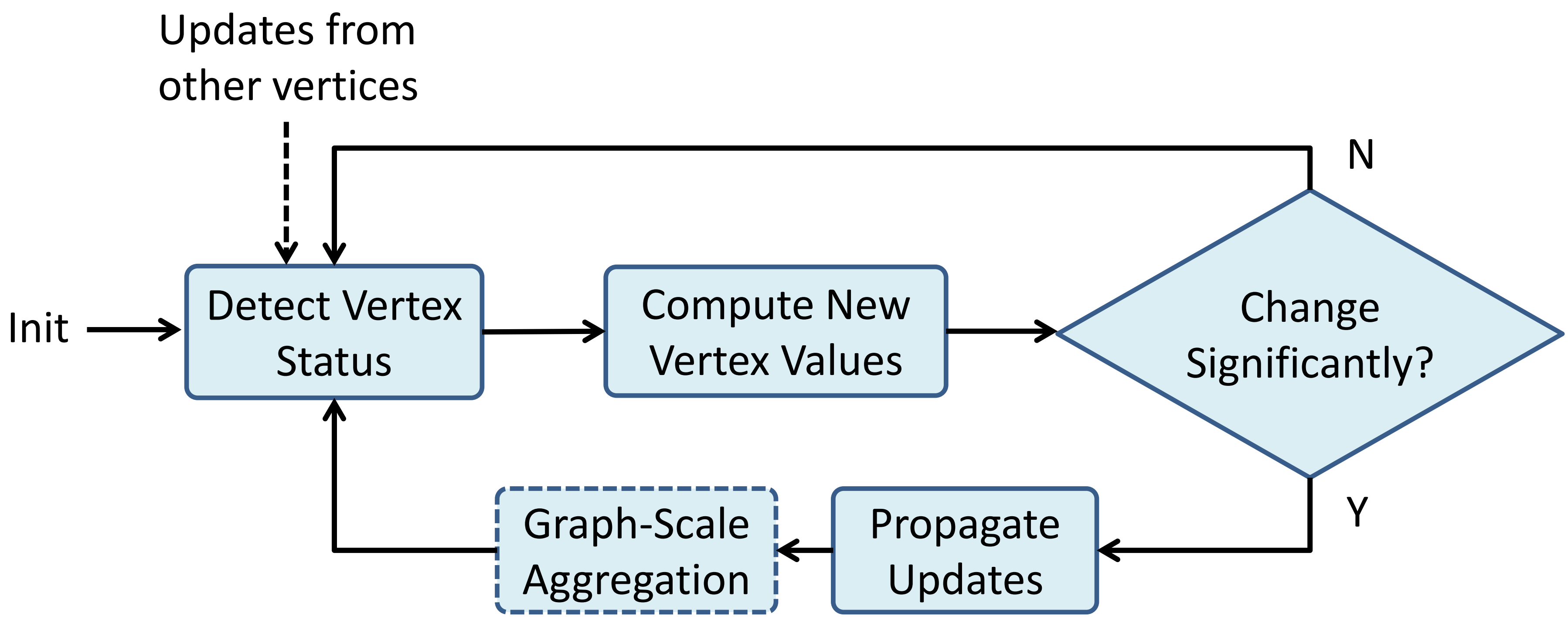
Epoch Commit Protocol



- No locking mechanisms required for global order
- Defer decisions to master snapshooter
  - Consensus on a set of ops and a *serializable* execution order
- Limitation: no cross-partition dependency across ops



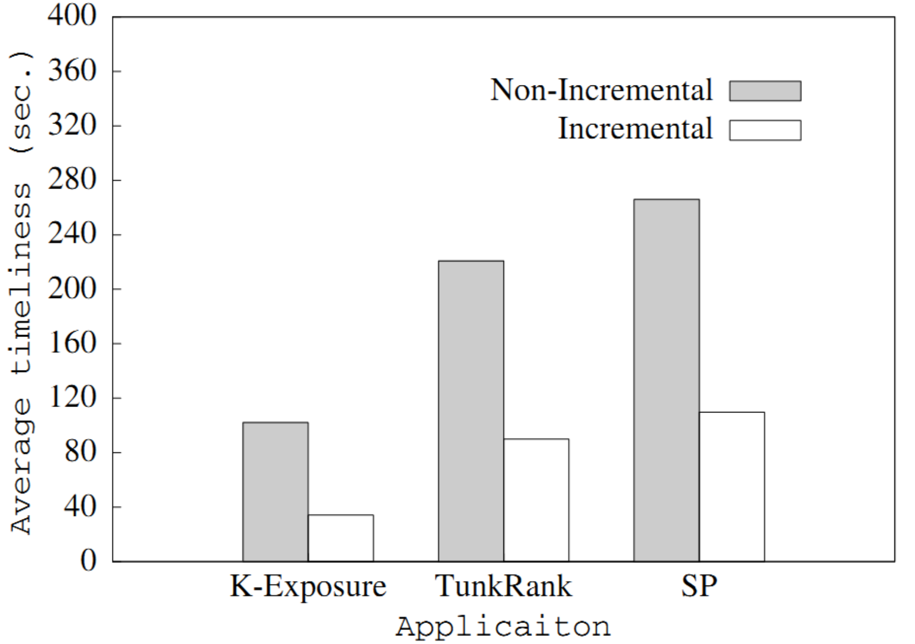
Incremental Graph Computation



Vertex-based iterative propagation

Selected Results

- Graph update rate
  - 180k tweet/s: 20x+ of Twitter peak record (Oct.2011)
- Incremental Computation



Contributions

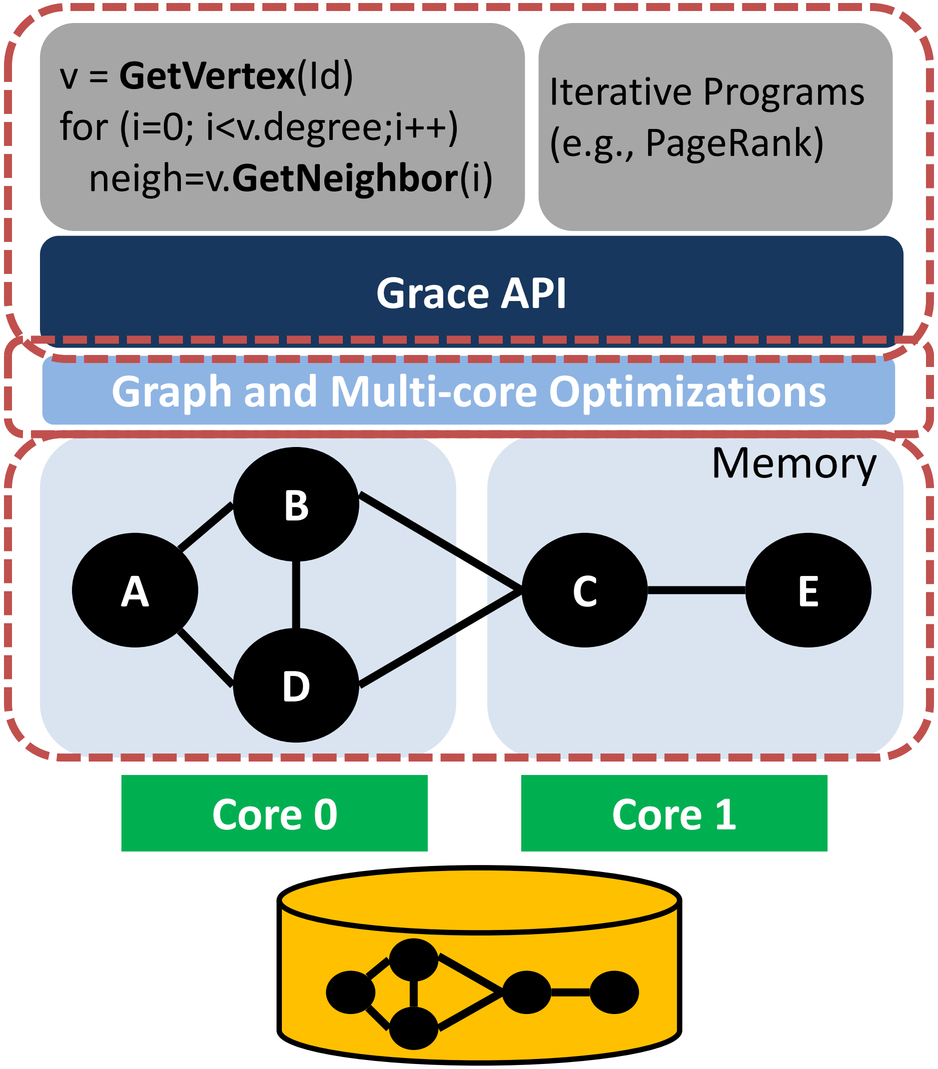
- **Kineograph**
  - A system that computes timely results on a fast changing graph
  - Separate graph update mechanism that supports high-throughput graph update and produces consistent snapshots
  - An efficient graph engine that supports incremental computation
- **Implementation validates design goals**
  - 100k+ sustainable update throughput and 2.5-minute timeliness with 40 machines

Grace

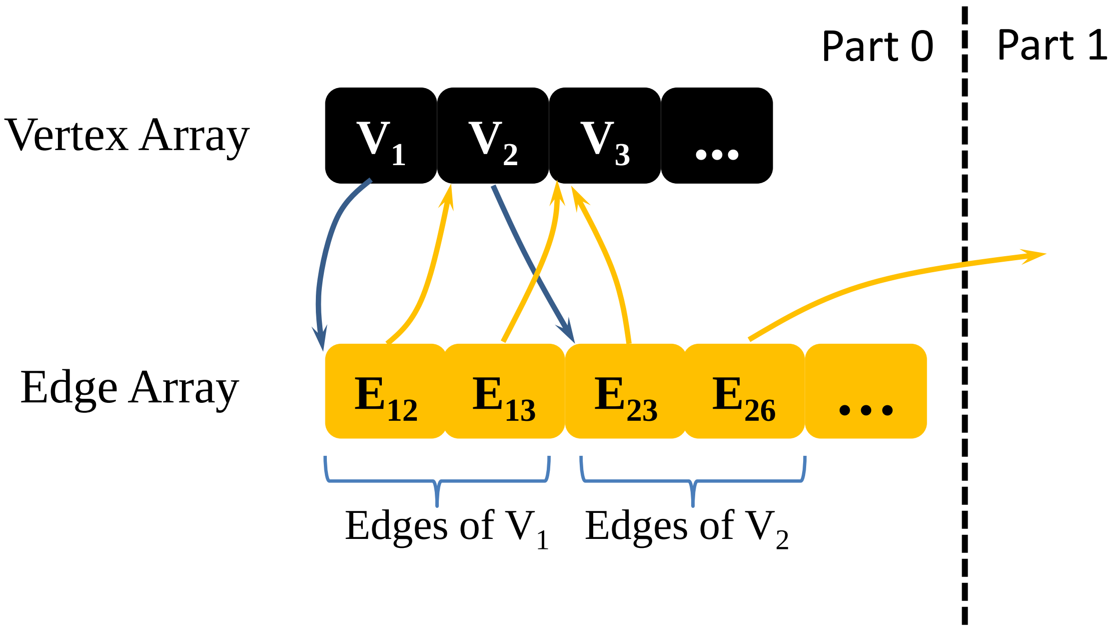
- A graph management and processing system
  - In-memory, single machine
  - Graph-specific and multicore-specific optimizations

- Orders of magnitude faster than existing systems
  - Berkeley DB, SQL-server, and Neo4j

An Overview of Grace



Graph-Aware Data Structures



Data Structures in a Partition

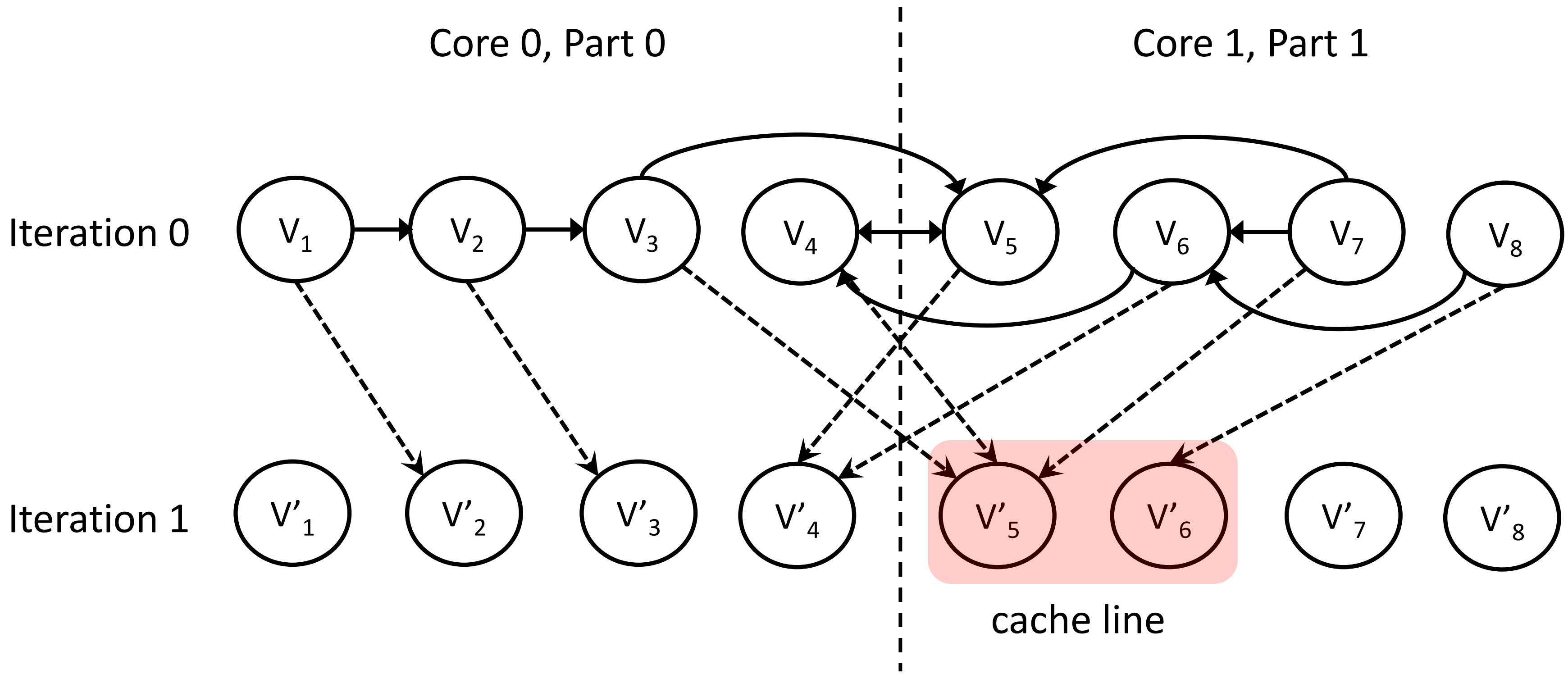
- Efficient, no indirect key-value lookup when following edges
- Enable graph-aware optimization on data locality

Graph-Aware Partitioning & Placement

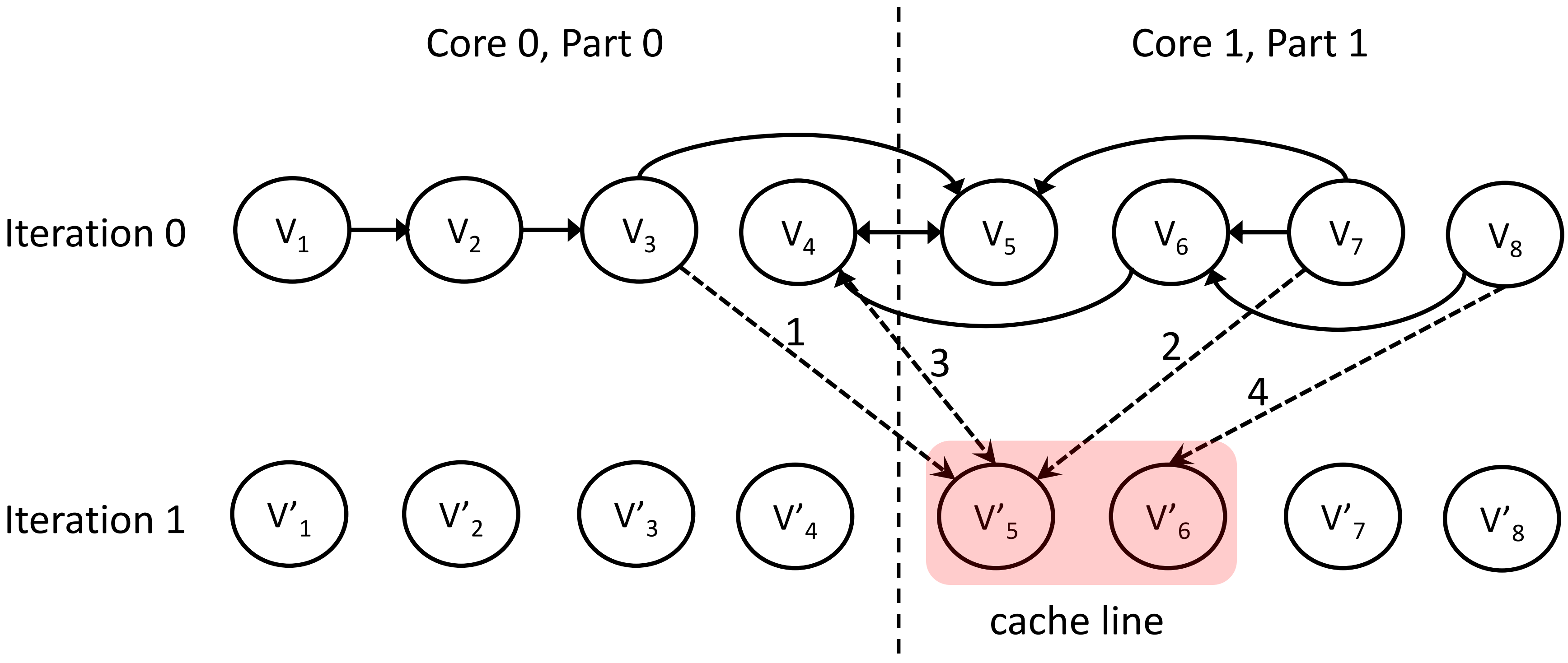
- Partitioning
  - Decrease cross-core communication & increase parallelism
  - Heuristic-based:
    - place v in a partition with more neighbors while balancing # of vertex across partitions, i.e., for each v, minimize $|Partition_i \setminus Neighbor_i(v)|$
  - Provides an extensible library
    - Metis, hash partitioning
- Placement
  - Better data locality: Place tightly connected vertices close
    - likely w/in one page and even CPU cache-line (during computation)
  - Spectral rearrange:
    - giving highly connected vertices similar score
    - arrange vertices in the order sorted by score

Platform for Parallel Iterative Computations

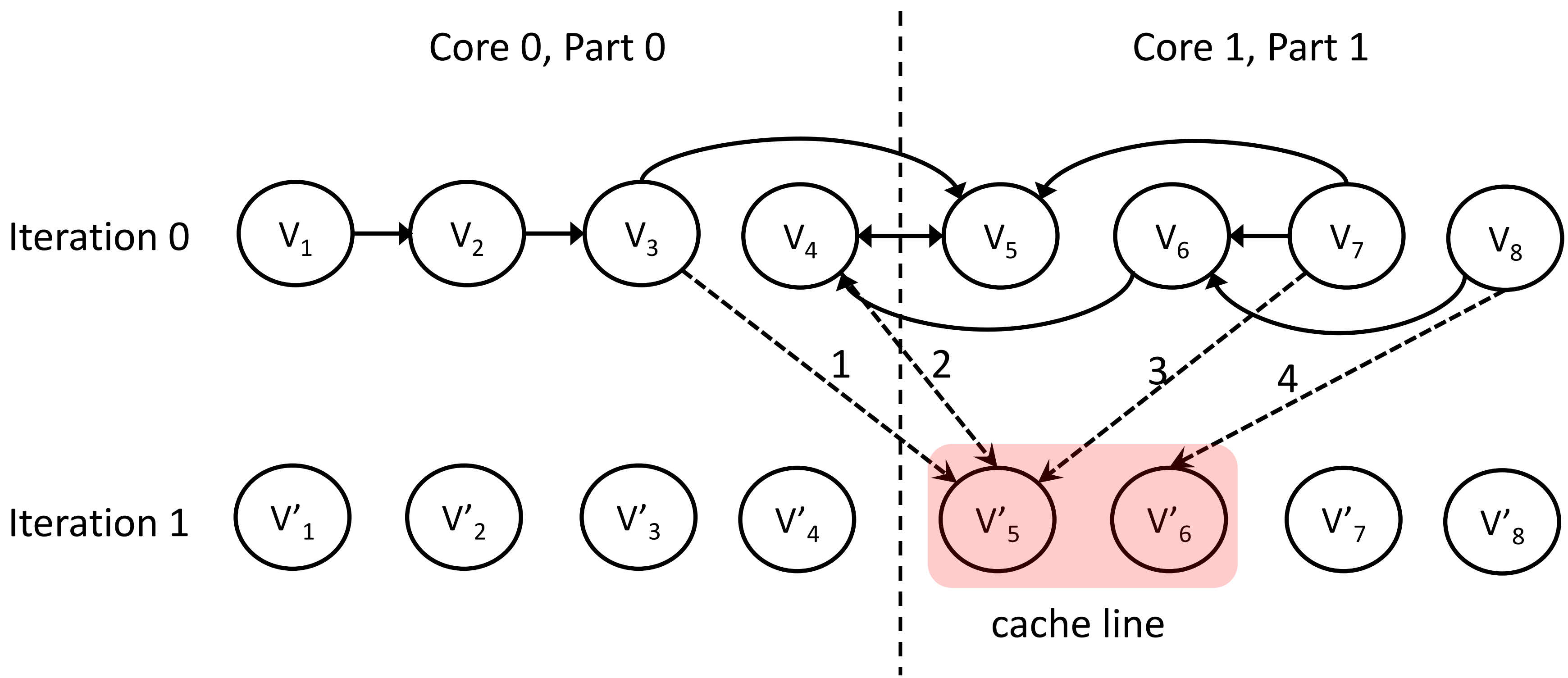
BSP (bulk synchronous parallel) model



Platform for Parallel Iterative Computations

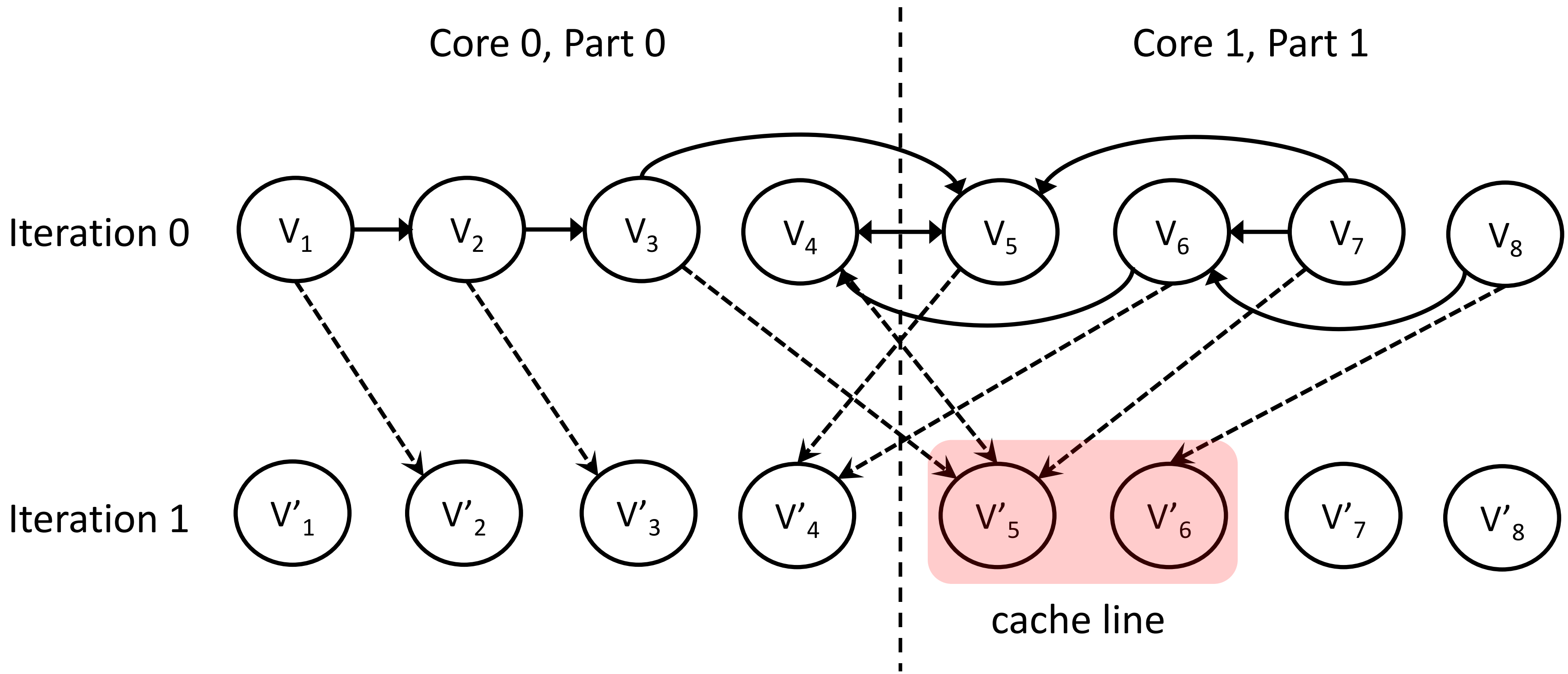


Platform for Parallel Iterative Computations

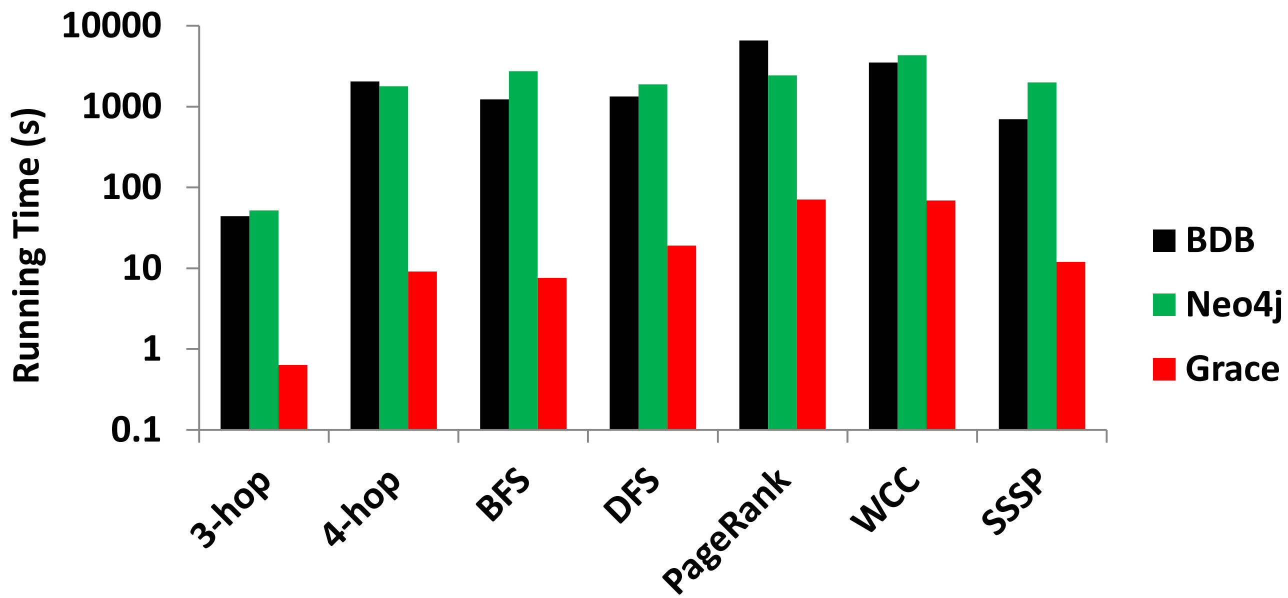


Platform for Parallel Iterative Computations

- Update Batching



Comparing Grace, BDB, and Neo4j



Orders of magnitude faster than existing alternatives

Conclusion

- Grace explores graph-specific and multi-core specific optimizations
- Careful vertex placement in memory gave good improvements
- Partitioning and updates batching worked in most cases, but not always

Backup

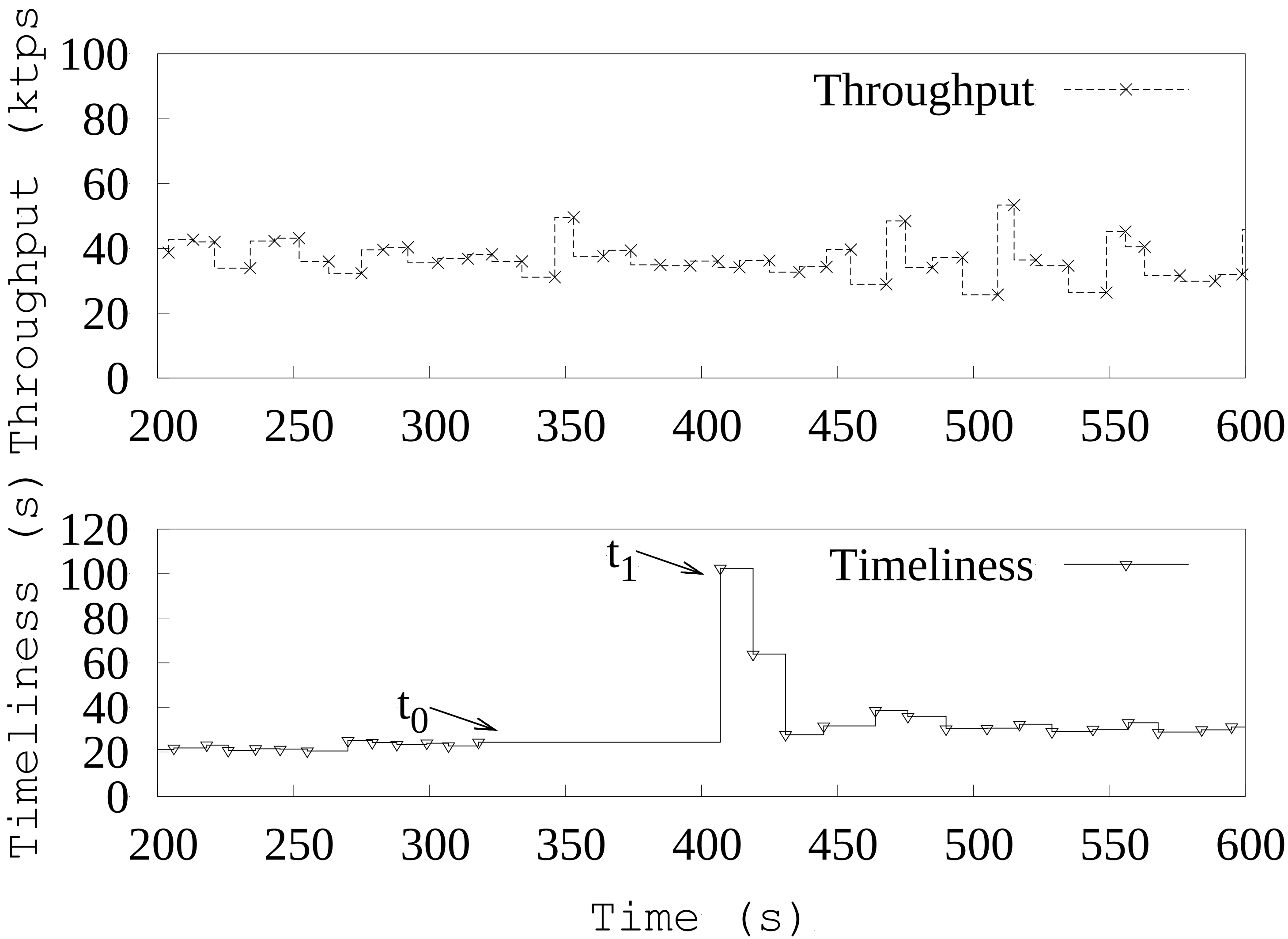
Kineograph Fault Tolerance

- Ingest node failure
  - Each ingest node *i* assigns an *incarnation* number along with each tx no. $[c_i, s_i]$ and marks it in the global progress table
  - A resurrected ingest node *i* seals c_i at s_i, and uses new incarnation number c_i+1: any op $[c_i, s]$ $(s > s_i)$ is discarded
- Graph node failure
  - Graph data : quorum-based replication, i.e., graph updates sent to k replicas and can tolerate f failures (k >= 2f+1)
  - No replication during computation: rollback and re-compute; computation results are replicated using primary backup
- Others: Paxos-based solution
  - Maintain progress table, coordinate computation, monitor machines, tracking replicas, etc.

Evaluation

- System implementation
  - Platform LoC: 16K~ C#
  - 3 Apps LoC: 1.5K~ C# (Influence Rank, approximate all-pair shortest path, hashtag-histogram)
  - 40+ servers, ~100M tweets
- Key performance numbers
  - Graph update rate: up to 180K tweets/s, 20+ times more than Twitter peak record (Oct.2011)
  - Influence Rank average timeliness over 8M vertices, 29M edges: ~2.5 minute

Failure Recovery



Programming with Kineograph

```
UpdateInfluence (v) { //event handling callback for a vertex
    val newRank = (1+p*v[“influence"]) / v.numOutEdges()
    foreach(e in vertex.outEdges()) {
      val oldRank = v.(”influence", e.target)
      val delta = |newRank – oldRank|
      if (delta > threshold)
         v.pushDeltaTo(“influence", e.target, delta)
   }  //pushDeltaTo propagates changes to other vertices
} //UpdateInfluence() triggered at changed vertices only
```

Snapshot Consistency

- Guarantee atomicity
  - All or none of the operations in a tx are included in a snapshot
- Global tx vector
  - A consensus on the set of tx to be included in a global snapshot
- Applying graph updates
  - Impose an artificial order within the set of tx: e.g., apply ops of s_1 first, and s_2, and so on.
  - Assumption: cross-partition ops do not have causal dependency

Applications

- Graph construction by extracting tweets
  - Mention graph: A @ B: A->B
  - HashTag graph: U posts a tweet that has #tagA: U->tagA
- Influence Rank: computing user influence
  - Calculate "PageRank" on a mention graph
- Approximate shortest paths
  - Shortest path between two vertices S(A,B): S(A, LandmarkA)+S(B, LandmarkB)
- K-Exposure: calculating hashtag exposure histogram (WWW'11)
  - If at time t user U posts a tweet S containing hash tag H, K(S) is the number of U's neighbors who post tweets containing H before t

Why focus on single machine?

- Single machine scale increases largely
  - Large main memory attached (10s~1000s GB)
  - Many cores (12~48, and even more)
  - Run workloads that are traditionally run on distributed systems
- Easy to deploy
  - No tricky distributed configurations
- Distributed graph system needs efficient local engine

Evaluation

Graphs:
- Web (v:88M, e:275M), sparse
- Orkut (v:3M, e:223M), dense

Workloads:
- N-hop-neighbor queries, BFS, DFS, PageRank, Weakly-Connected Components, Shortest Path

Architecture:
- Intel Xeon-12 cores, 2 chips with 6 cores each
- AMD Opteron-48 cores, 4 chips with 12 cores each

Questions:
- How well partitioning and placement work?
- How useful are load balancing and updates batching?
- How does Grace compare to other systems?