Pregel

A System for Large-scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski

Presentation by: Jesse Yang

Maximum Value Vertex in a Graph

Given a directed graph, how might you find the maximum value across all vertices?

- What if the graph was very large?
- What if the problem was more complex?

Pregel provides a system and way of thinking to tackle graph problems like these in parallel.

Existing Parallel Graph Systems

Prior to Pregel, libraries like Parallel BGL and CGMgraph existed.

- Libraries support selection of distributed graph algorithms
- Libraries are limited by implemented algorithms and their implementations
- Don't address fault tolerance and other issues large graphs can be subject to in distributed computing

Large Graphs

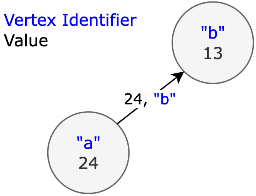
- Large graphs are often used for potent computing problems, e.g. Web graph, social networks, etc.
- Efficient processing is especially difficult due to their size
- Prior to the paper, run algorithms on large graphs via:
  - Create a new distributed infrastructure for particular problem (high effort)
  - Use existing distributed computing platform, e.g. MapReduce (unfit)
  - Single-computer graph algorithm libraries (limited scope)
  - Existing parallel graph systems, e.g. Parallel BGL (limited and not fault-tolerant)
- Pregel is a *system* that is scalable, flexible, and fault-tolerant

Table of Contents

Table of Contents

Setup and Structure

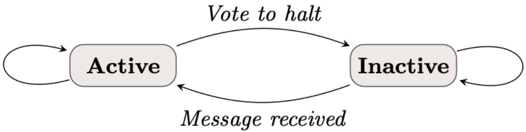
- Input is a *directed* graph
  - Each vertex has a vertex identifier and modifiable associated value
  - Each directed edge is associated with a source vertex, and has a modifiable value and target vertex identifier



Superstep

- Computations involve sequence of iterations, **supersteps**
- Within a superstep S, vertices compute conceptually in parallel (vertex-centric system):
  - Can modify the state of itself or outgoing edges
  - Can receive messages sent from vertices in $S-1$ and send messages to vertices to be received in $S+1$
  - Can modify the graph's topology

Termination



- All vertices start *active* but can vote to halt
- Inactive vertices are reactivated upon receiving a message
- Algorithm terminates when all vertices are inactive
- Output of algorithm is set of values output by vertices
- Message passing amortizes latency by batching (remote reads unnecessary)

Maximum Example

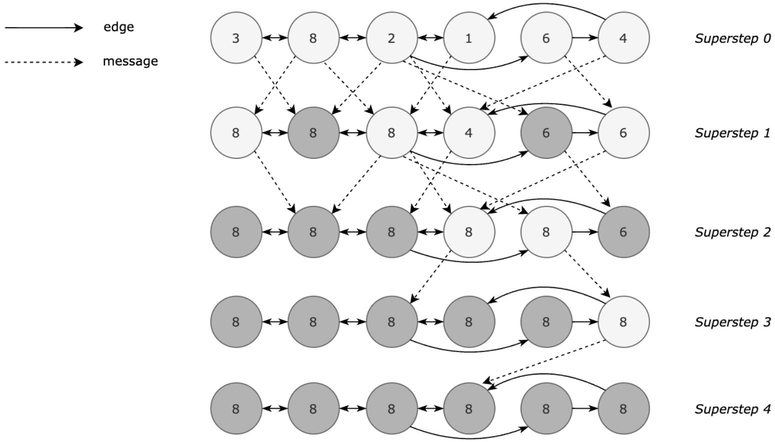


Table of Contents

Architecture Overview

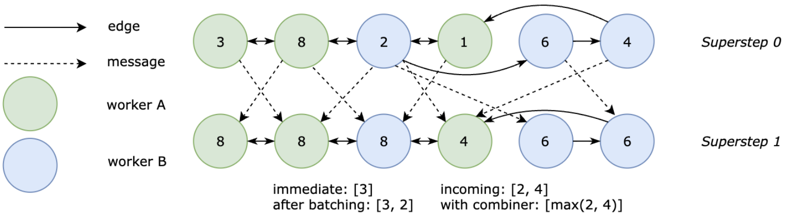
Graph divided into partitions, containing vertices and all their outgoing edges (default by hash)

1. Copies of user program execute on cluster of machines, with one copy designated and known by others as master
2. Master determines partitions of graph, assigning partition(s) to workers
3. Master portions user input across workers to initialize
4. As long as active vertices, master instructs workers to perform superstep (one thread per partition); worker responds with number of active vertices next superstep

Worker Implementation

- Worker machines maintain state of their portion of the graph
  - Vertex info: value, outgoing edges (value and target vertex), incoming messages, active flag
- Superstep by looping through all vertices and calling `Compute()`
  - Passes current value, iterator to incoming messages (no guaranteed order), iterator to outgoing edges
  - Same function executed at all vertices
  - Maintain two copies of active vertex flags and incoming message queue
- Handle sending messages to other vertices (batch remote and immediate local)
  - User-defined **combiners** can combine messages via commutative and associative operations, e.g. addition

Worker Implementation (Maximum Example)



Worker Implementation (Topology Mutations)

- `Compute()` can issue requests to add/remove vertices/edges
  - Mutations take effect in next superstep
- Partial ordering in case of conflicts:
  1. Edge removal
  2. Vertex removal
  3. Vertex addition
  4. Edge addition
- Final resort: independent user-defined handlers (keeping `Compute()` simple)

Master Implementation

- Maintain alive workers, addressing information, and assigned graph portions
  - Size required proportional to number of partitions
- Coordinate worker activity
  - Same request sent to each worker
- Maintain stats on computation and state of graph

Aggregators

- **Aggregators** allow global monitoring, data, and communication
- Aggregators can take vertex-provided values in superstep S, combine using a reduction operator, made available at $S + 1$
- Useful for stats (total # edges), global coordination (synchronized branching)
- Workers maintain aggregator instances, partially reduce over vertices, tree-based reduction across workers delivered to master

Fault Tolerance

- Master instructs workers to save state to persistent storage at beginning of superstep
- Failures detected via "ping" messages issued from master to worker
- **Confined recovery** limits recovery to lost partitions

Table of Contents

PageRank

Goal: Roughly estimates how important a page (vertex) is based on links (edges) to it

- Vertex values v: tentative page rank (initialized to $1/n$, n vertices)
- Outgoing message: v
- `Compute()`: $v \leftarrow 0.15/n + 0.85 \cdot \sum_{u \in \text{incoming}} msg[u]$
- Run until some level of convergence ϵ

Single-Source Shortest Paths

Goal: Finds the shortest distance from a source vertex s to every other vertex in the graph

- Vertex values v: current shortest path from s (initialized to 0 for s, ∞ otherwise)
- Outgoing message for edge e: $v + e.v$ (potentially new shortest distance)
- **`Compute()`**: $v \leftarrow \min(\min_{u \in \text{incoming}} msg[u], v)$
- Run until no more updates (termination guaranteed for nonnegative edge weights)
- Can use a minimum combiner

Table of Contents

SSSP Scale With Worker Tasks

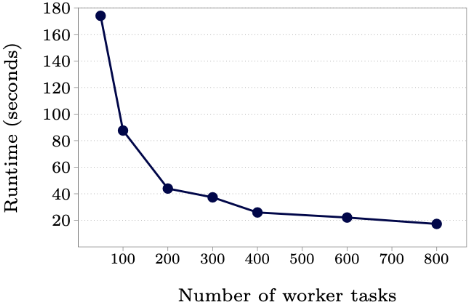


Figure 7: SSSP—1 billion vertex binary tree: varying number of worker tasks scheduled on 300 multicore machines

SSSP Scale With Graph Size

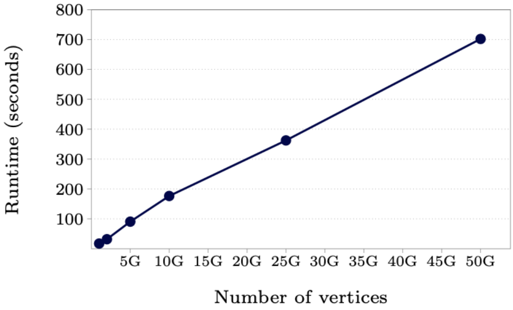


Figure 8: SSSP—binary trees: varying graph sizes on 800 worker tasks scheduled on 300 multicore machines

SSSP on Log Normal Random Graphs

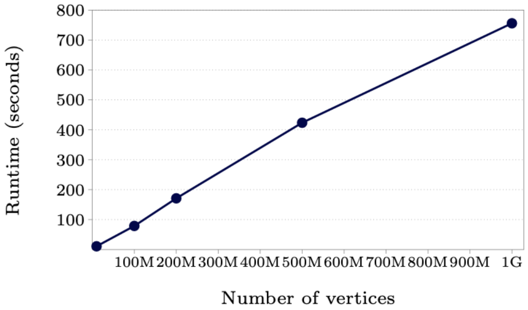


Figure 9: SSSP—log-normal random graphs, mean out-degree 127.1 (thus over 127 billion edges in the largest case): varying graph sizes on 800 worker tasks scheduled on 300 multicore machines

Notes on Results

- Topology-aware partitioning would perform better
- More advanced algorithm would perform better
- Results merely indicate satisfactory performance (comparable to Parallel BGL and scales better)
- Mainly designed for sparse graphs where communication primarily resides over edges

Considerations

- Master operations require barrier synchronization
  - Faster workers frequently have to wait to synchronize between supersteps
- Serializability not provided due to delay of supersteps