Pregel

Ali Shah
s9alshah@stud.uni-saarland.de

Outline

- Introduction
- Model of Computation
- Fundamentals of Pregel Program
- Implementation
- Applications
- Experiments
- Issues with Pregel

Outline

- Costs of Computation
- Optimization Techniques and Algorithms
- Experiments
- Criticism
- Conclusion

Introduction

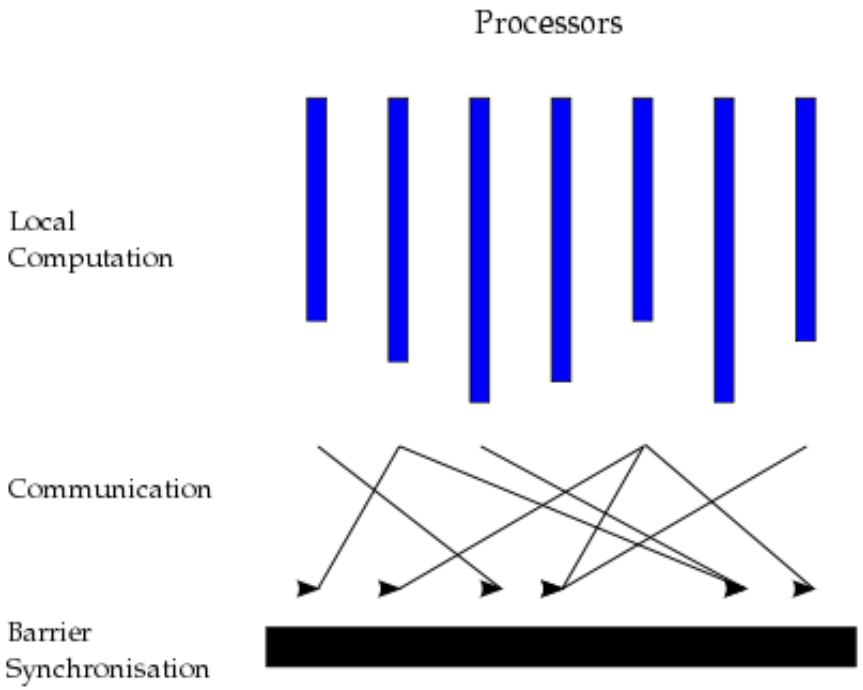
- Analyzing large graph is hard
  - Billions of edges
  - Trillions of vertices
  - Examples: Web graph, Social Networks, Transportation Networks
- Graph algorithms
  - No scalable general-purpose system for implementing arbitrary graph algorithms

Pregel, to the Rescue

- Framework for processing large graphs
- Easy to program
- Scalable
- Fault Tolerant
- Inspired by Bulk Synchronous Parallel Model
- Can implement most of the graph algorithms
- Vertex-centric system

Model of Computation

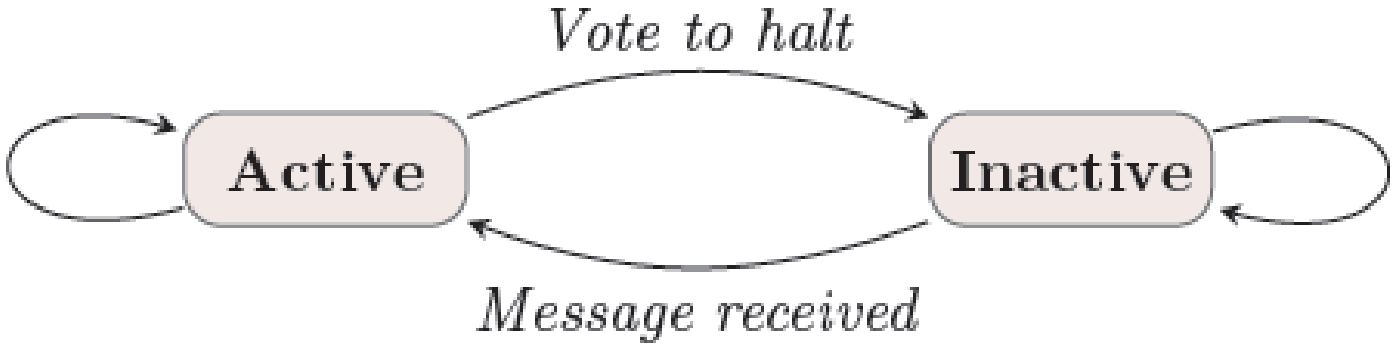
- Sequence of iterations (supersteps)
  - Same function (user-defined) is executed for each vertex



Source: http://blog.acaro.org/entry/pregel-is-out-but-what-is-pregel

Model of Computation

Vertex State Machine



- Algorithm terminates when all vertices are simultaneously inactive and there are no messages in transit

Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Model of Computation

- Vertex
  - Receives messages sent in previous superstep
  - Executes the user defined compute function
  - Updates its or its outgoing edges' values
  - Sends messages to other vertices
  - Update the graph structure
  - Votes to halt if done with the task

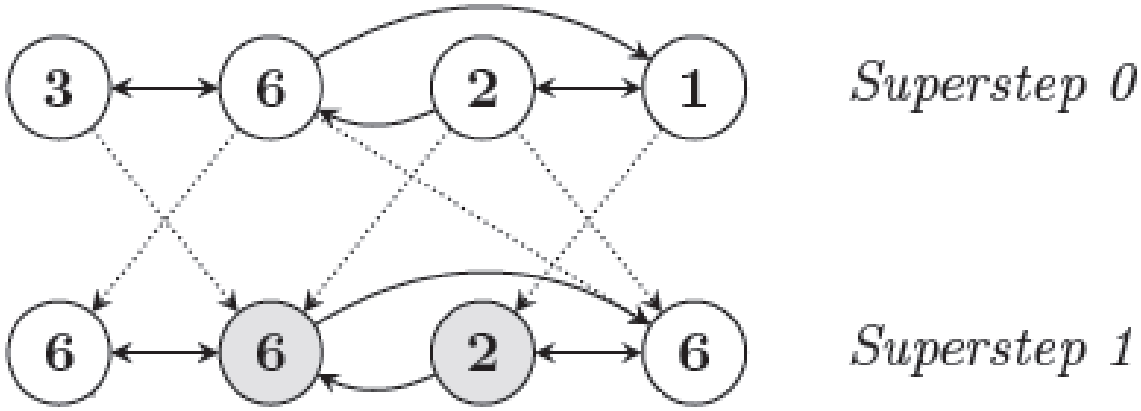
- Communication is done through message passing
- Concurrent computation and the messages need not be ordered

Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Example: Maximum Value

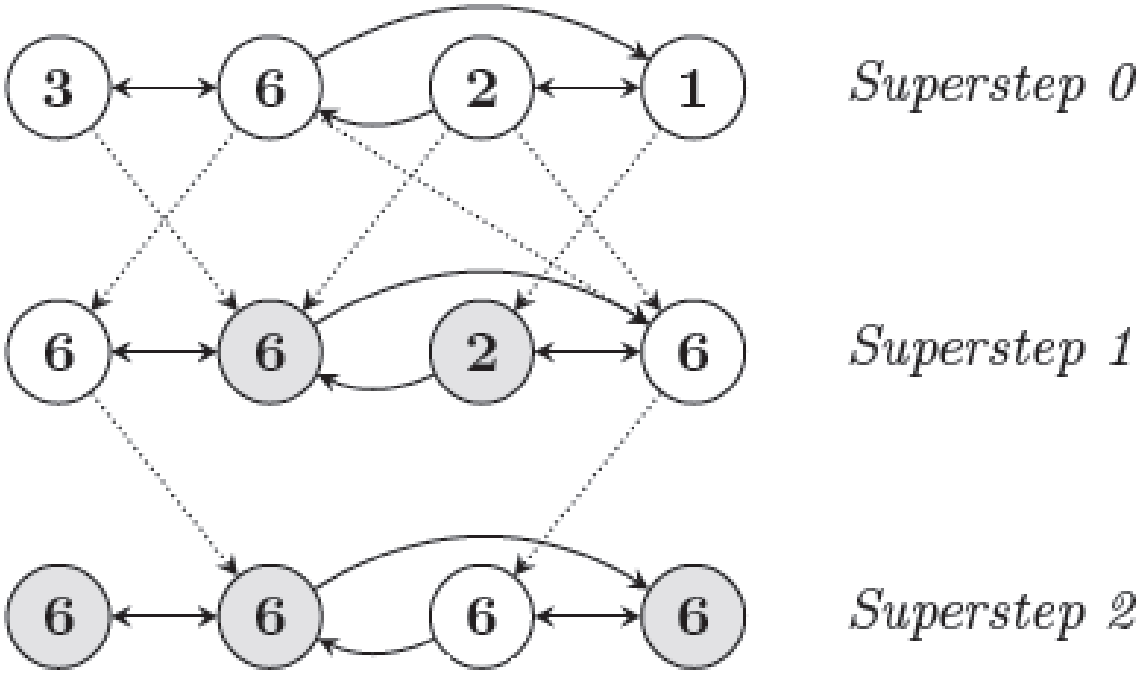
Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Example: Maximum Value



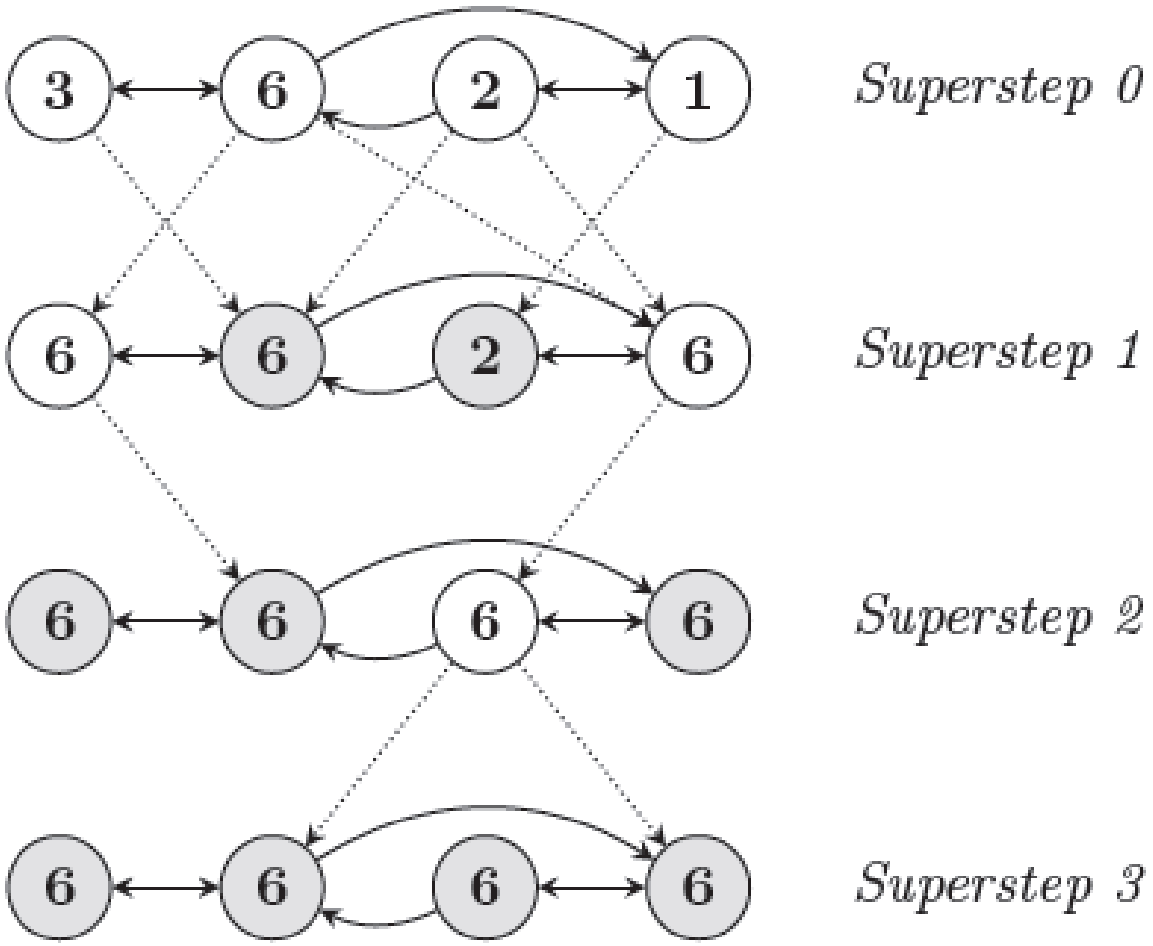
Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Example: Maximum Value



Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Example: Maximum Value



Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Pregel vs. Map Reduce

- Pregel keeps vertices and edges on the machine that performs computation
- Pregel uses network transfer only for messages

- Map Reduce passes the entire graph state from one state to the next
- Map Reduce needs to coordinate chained steps

Fundamentals of Pregel Program

- Vertex.compute() function
- Combiners
- Aggregators

Fundamentals of Pregel Program: Vertex.Compute function

- Receive messages that were sent in previous superstep
- Update vertex and edge values
- Send messages
- Update aggregators
- Choose to vote to halt

Fundamentals of Pregel Program: Combiners

- Example
  - Integer messages received
  - Only sum matters
- Messages can be combined
- Reduces the number of messages that must be transmitted
- Should only be enabled for commutative and associative operations

Fundamentals of Pregel Program: Aggregators

- Mechanism for global communication
- Each vertex can provide value in superstep S
- System combines these values using a reduction operator
- Resulting value made available to all vertices in superstep S+1
- Can be used for statistics
- Example:
  - Total number of edges in graph – Sum aggregator on out-degree of each vertex

Implementation

- Basic Architecture
- Execution of Pregel Program
- Fault Tolerance

Implementation: Basic Architecture

- Graph is divided into partitions
  - Default partitioning function is hash(VertexId)
  - User can provide the custom partitioning function
- Master Worker model
  - One master, multiple workers
  - Master's tasks
    - Maintenance of workers
    - Fault recovery of workers
    - Web UI for status tracking
  - Workers' tasks
    - Processing of assigned tasks
    - Communication with other workers (message passing)

Implementation: Execution of Pregel Program

- Program begins executing on a cluster of machines
- Master is responsible for coordinating worker activity
- Master partitions the graph and assigns one or more partitions to each worker
- All vertices are marked as active

Implementation: Execution of Pregel Program

- Master instructs each worker to perform a superstep.
- Worker
  - Calls Compute() function for each vertex
  - Receives messages that were sent in previous superstep
  - Sends the messages
  - Tells the master how many vertices will be active in the next superstep
  - Repeat while any vertices are active or any messages are in transit
- After the computation halts, master may instruct each worker to save its portion of the graph

Implementation: Fault Tolerance

- Checkpointing
  - Workers save their state of partition on persistent storage
- Failure detection
  - Ping messages to workers
- Recovery of workers
  - Assignment of graph partition to available worker
  - Workers reload state from the last checkpoint and continue execution

Applications

- Shortest Paths
- Bipartite Matching
- Page Rank
- Semi-Clustering

Application: Shortest Paths

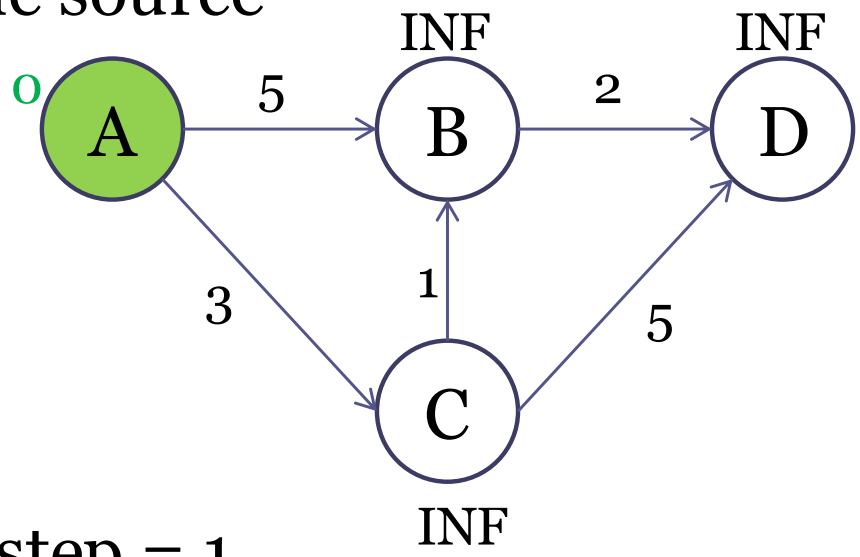
- Objective
  - Single source shortest paths: Finding shortest path between a single source vertex and every other vertex in the graph
  - s-t shortest path: Finding a single shortest path between given vertices s and t

Application: Single Source Shortest Paths

- Each vertex stores a value denoting the distance from source vertex to this vertex
- Value at each vertex is initialized to INF
- In each superstep
  - Receives messages from its neighbors with updated potential minimum distances from source vertex
  - If minimum of these updated values is less than the current minimum distance of the vertex, value is updated and potential updates are sent to the neighbors (current value + outgoing edge weight)
- In first superstep, only source vertex will update its value to zero and send update messages to its neighbors
- Algorithm terminates when no more updates

Application: Single Source Shortest Paths

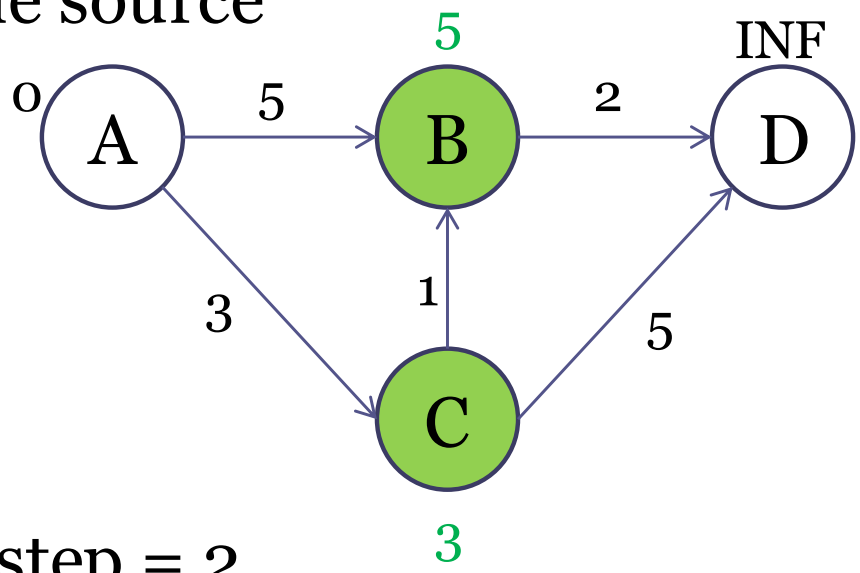
- A is the source



- Superstep = 1
  - A = 0
  - A sends messages
    - B = 0+5 = 5
    - C = 0+3 = 3

Application: Single Source Shortest Paths

- A is the source



- Superstep = 2
  - B = 5; C = 3
  - B sends messages
    - D = 5+2 = 7

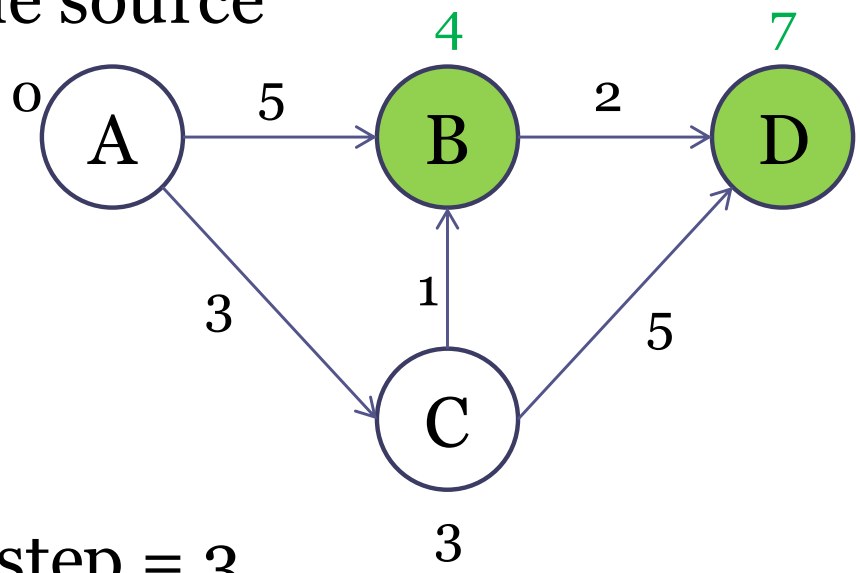
C sends messages

B = 3+1 = 4

D = 3+5 = 8

Application: Single Source Shortest Paths

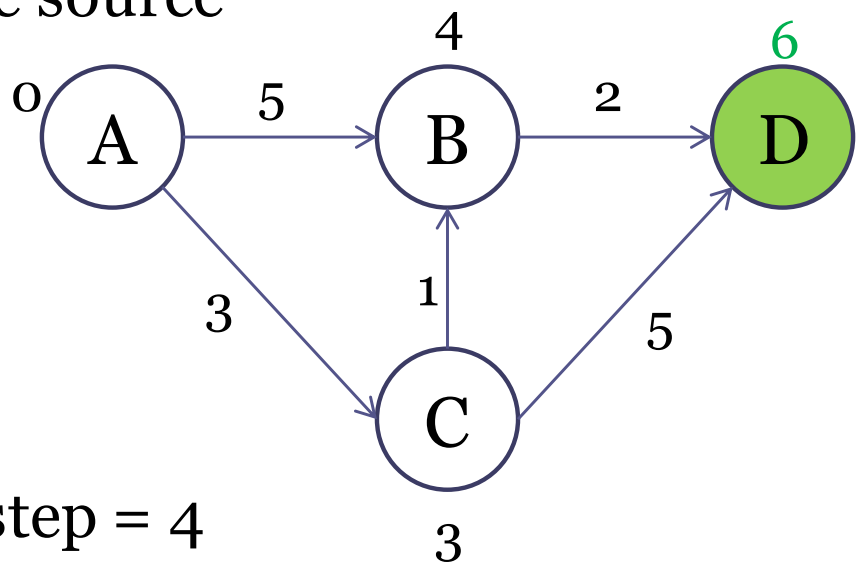
- A is the source



- Superstep = 3
  - B = 4; D = 7
  - B sends messages
    - D = 4+2 = 6

Application: Single Source Shortest Paths

- A is the source

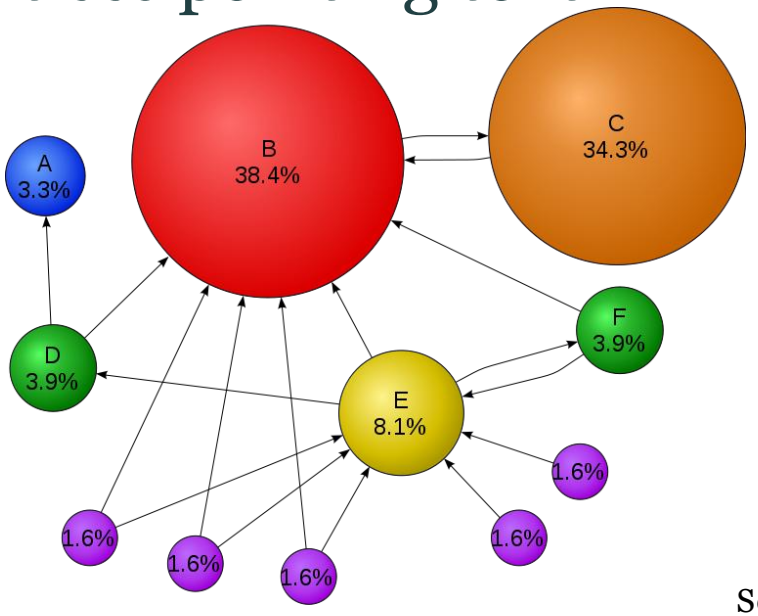


- Superstep = 4
  - D = 6

- Since there will be no incoming messages in next step, the algorithm will terminate
- Values at vertices are the shortest distance from the source

Application: PageRank

- Objective
  - Method to measure the importance of the vertices in graph
  - Importance of the vertex depends upon the count and quality of vertices pointing to it



Source: http://en.wikipedia.org/wiki/PageRank

Application: PageRank

- In superstep 0, the value of each vertex is 1/NumVertices()
- In each superstep
  - Each vertex sends along each outgoing edge its tentative PageRank divided by the number of outgoing edges
  - Each vertex sums up the values received in the messages
  - Tentative PageRank for the vertex is updated to *0.15/NumVertices* + *0.85* x *sum*
- Algorithm termitates on convergence

Application: PageRank

```
class PageRankVertex
    : public Vertex<double, void, double> {
 public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() =
          0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
};
```

Calculating sum using incoming edges

Tentative PageRank

Sending messages to outgoing edges

Halts at superstep = 30

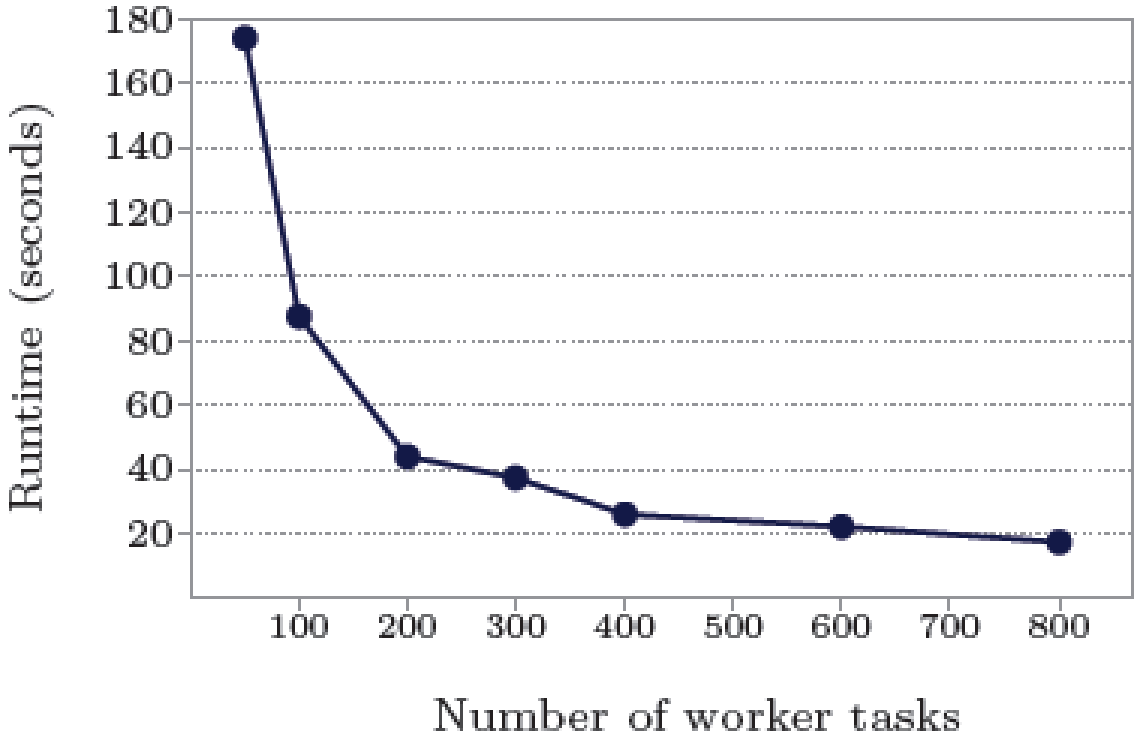
Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Experiments: Environment

- 300 multicore commodity PCs
- 50 to 800 pregel workers
- Weights of all edges set to 1
- Graph
  - Binary Tree
  - Random Graphs

Experiments

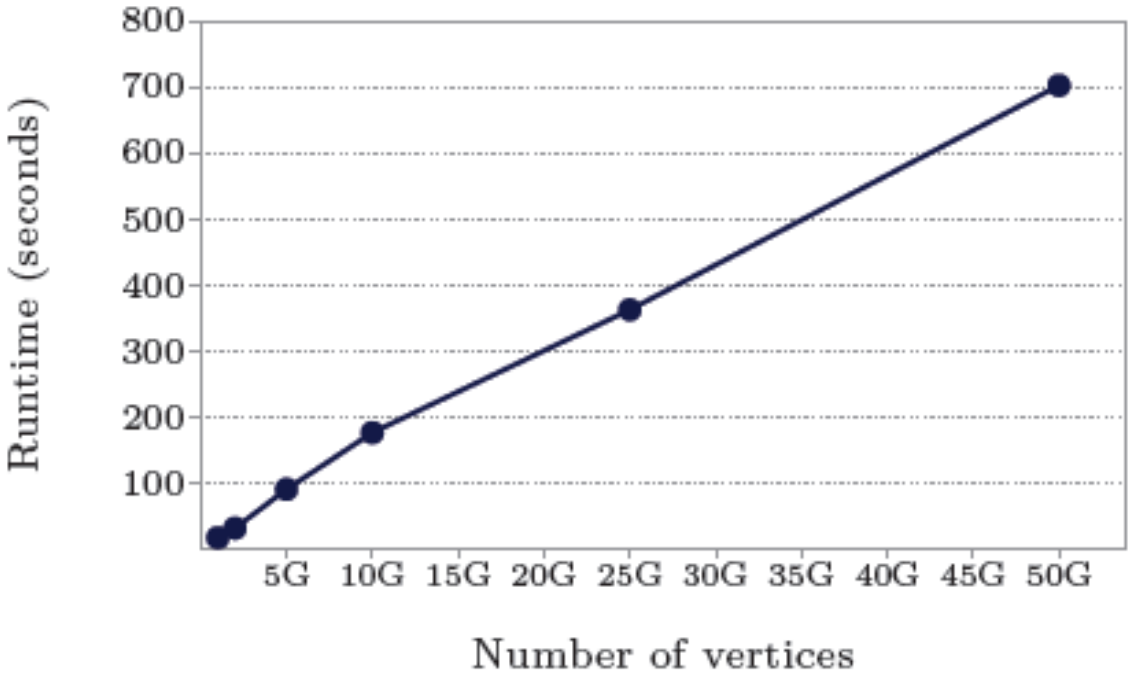
- Single Source Shortest Paths – Binary tree with 1 billion vertices – Number of worker tasks vary



Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Experiments

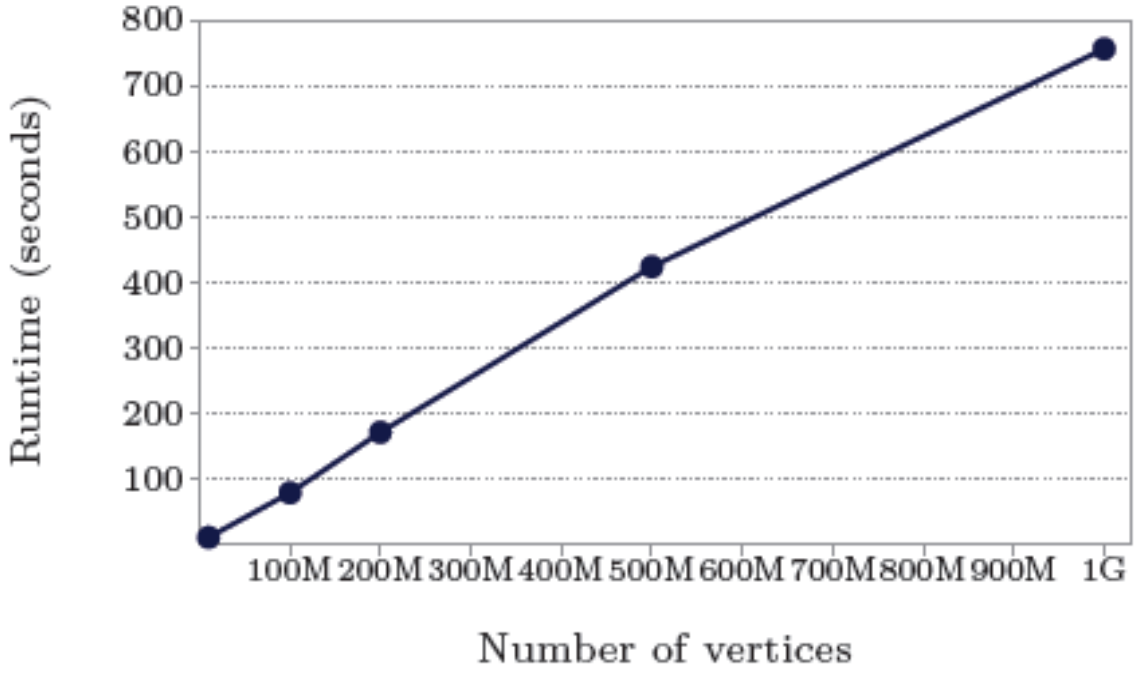
- Single Source Shortest Paths – Binary tree – 800 Worker tasks – Graph size vary



Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Experiments

- Single Source Shortest Paths – Log normal random graphs with 127 billion edges – 800 Worker tasks – Graph size vary



Source: https://kowshik.github.io/JPregel/pregel_paper.pdf

Pregel in a Nutshell

- Vertex centric approach
- Concept of supersteps
- Master.Compute function
- Massage passing between vertices
- Open source implementations present
  - Giraph
  - GPS

Issues with Pregel

- Convergence is slow
- High communication or computation cost
  - Graphs with skews in component sizes

Costs of Computation

- Four different costs
  - Communication
  - Number of supersteps
  - Memory
  - Computation by each vertex in each superstep

- Optimization techniques focus on the first two

Optimization Techniques

- Finish Computations Serially
- Storing Edges at Subvertices
- Edge Cleaning on Demand
- Single Pivot Optimization

Optimization: Finish Computations Serially

- Motivation
  - Slow convergence in an algorithm or phase of an algorithm (Execution of large number of supersteps when working on very small fraction of input graph)
  - Communication cost degrades performance in such cases
- Optimization
  - Avoids large number of small superstep executions by finishing computation on a small active-subgraph serially, inside master.compute()
  - Can be applied to algorithms in which size of active subgraph shrinks throughout the computation

Optimization: Finish Computations Serially

- Implementation
  - Uses three global objects
    - Number of edges in active-subgraph
    - Active subgraph when serial computation is triggered
    - Results of the serial execution
  - The serial computation is performed inside master.compute()

Optimization: Finish Computations Serially

- Cost Analysis
  - Avoid additional superstep executions
  - Overhead
    - Monitoring size of active-subgraph
    - Serial computation at the master
    - Communication cost of sending active-subgraph to the master and results back to the workers
    - One superstep for vertices to read the results
- Optimization is expected to yield good benefits only when algorithm or the phase of algorithm converges very slowly

Optimization: Finish Computations Serially

- Example: Graph Coloring
  - Objective
    - Assigning a color to each vertex such that no two adjacent vertices have the same color

Optimization: Finish Computations Serially

- Example: Graph Coloring
  - Procedure
    - Each vertex sets its type to unknown (not yet decided)
    - Sends message to its neighbors for degree calculation
    - Vertex, with 1/(2 x degree(vertex)) probability, volunteers to be in maximal independent set
      - Sends messages to all its neighbors
    - Each vertex that had volunteered, checks the messages it has received. If its Id is minimum, becomes part of maximal set. Sends "neighbor-in-set" message to its neighbors
    - Vertices that receive this message update their type to NotInS, send "decrement degree" message to its neighbors, and becomes inactive
    - Vertices receiving this message update their degree count
    - If further unknown vertices left, process is repeated, otherwise maximal independent set has been generated and a color is assigned to it.

Optimization: Finish Computations Serially

- Example: Graph Coloring
  - Optimization
    - Over the time, active-subgraph gets denser, as a result independent sets get smaller.
    - Can be left with a small clique producing as many independent sets as the vertices in clique
    - If active subgraphs become smaller than a threshold, task is executed serially by the master, saving some supersteps

- Similar optimizations can be done on Strongly connected components algorithms in which strongly connected components are to be found from the graph

Optimization: Storing Edges at Subvertices

- Motivation
  - Algorithms in which supervertices is formed (Eg. Minimum Spanning Forest)
    - Subvertices are merged to form supervertices
  - High cost for receiving and merging adjacency list of subvertices

- Optimization
  - Store edges of supervertex in distributed fashion among all of its subvertices

Optimization: Storing Edges at Subvertices

- Implementation and Example: Minimum Spanning Forest
  - Objective
    - Minimum Spanning Forest is a collection of Minimum Spanning Trees that connect the vertices of the graph together

Optimization: Storing Edges at Subvertices

- Implementation and Example: Minimum Spanning Forest
  - Procedure
    - Each vertex selects its minimum weight edge
    - Each vertex sends a message to the vertex at other end of the selected edge. By this supervertex and cycle in the conjoined tree is identified.
    - Edge Cleaning and Relabeling
      - Each vertex sends its and its supervertex's Id to all of its neighbors.
      - When messages are received in next super step, if vertices share same supervertex, the edge between them is deleted. Else, current vertex's supervertex is updated to point at the updated supervertex (received in the message)
    - Supervertex Formation
      - Every subvertex sends its edge to its supervertex
      - Each supervertex merges and stores these edges

Optimization: Storing Edges at Subvertices

- Implementation and Example: Minimum Spanning Forest
  - Optimization
    - Supervertex formation is a high cost operation as every subvertex sends its edges to supervertex and then supervertex merges these list from subvertices
    - Storing edges of supervertex in distributed fashion among all subvertices.
    - Subvertices send its ID to the supervertex
    - Supervertex sends messages back to its subvertices with new supervertex ID
    - Subvertices update their supervertices with the ID received

Optimization: Storing Edges at Subvertices

- Cost Analysis
  - The computation and communication performed in the supervertex formation phase is avoided
    - Cost is proportional to the number of edges in the graph
  - Additional communication cost
    - Subvertex sends local minimum weight edges to supervertex
    - Subvertex sends its Id to supervertex
    - Supervertex sends messages to subvertex with updated supervertex Id
- Overall increase or decrease in communication depends upon the sizes of active vertices and edges

Optimization: Edge Cleaning on Demand

- Motivation
  - Edge Cleaning: Removing edges based on certain conditions (vertex values)
  - Implementation in Pregel: Vertices send messages to their neighbors in one superstep, and remove neighbors in another
  - Communication cost proportional to the number of edges
- Optimization
  - Keep stale edges around instead of deleting them
  - Stale edge deleted only when vertex tries to use it as a part of computation

Optimization: Edge Cleaning on Demand

- Implementation and Example: Minimum Spanning Forest
  - Procedure
    - Each vertex selects its minimum weight edge
    - Each vertex sends a message to the vertex at other end of the selected edge. By this supervertex and cycle in the conjoined tree is identified.
    - Edge Cleaning and Relabeling
      - Each vertex sends its and its supervertex's Id to all of its neighbors.
      - When messages are received in next super step, if vertices share same supervertex, the edge between them is deleted. Else, current vertex's supervertex is updated to point at the updated supervertex (received in the message)
    - Supervertex Formation
      - Every subvertex sends its edge to its supervertex
      - Each supervertex merges and stores these edges

Optimization: Edge Cleaning on Demand

- Implementation and Example: Minimum Spanning Forest
  - Procedure
    - Each vertex selects its minimum weight edge
    - Each vertex sends a message to the vertex at other end of the selected edge. By this supervertex and cycle in the conjoined tree is identified.
    - **Edge Cleaning and Relabeling**
      - Each vertex sends its and its supervertex's Id to all of its neighbors.
      - When messages are received in next super step, if vertices share same supervertex, the **edge between them is deleted**. Else, current vertex's supervertex is updated to point at the updated supervertex (received in the message)
    - Supervertex Formation
      - Every subvertex sends its edge to its supervertex
      - Each supervertex merges and stores these edges

Optimization: Edge Cleaning on Demand

- Optimization
  - Edge cleaning and relabeling phase is omitted completed
  - Now the vertices cannot discover their minimum edge as some of the edges may be stale
  - Additional phase – Stale-Edge-Discovery
    - Vertex v sends message to minimum-weight edge(v,u) with its ID and its supervertex ID.
    - If u belongs to different supervertex, it answers back with its supervertex ID.
    - If v receives an answer message, it picks u as its minimum-weight edge and updates its supervertex ID
    - If v does not receive an answer, it removes u
- Similar optimization can be perfomed on Maximum Weight Maching Algorithm as well in which matchings in the graph is to be found for which the sum of the weights of the matched edges is as large as possible

Optimization: Edge Cleaning on Demand

- Cost Analysis
  - Reduces the cost of communication and computation of sending messages for deleting some edges
  - May slow down the convergence of the algorithm, decreasing the number of vertices that match, which increases the number of iterations

Optimization: Single Pivot

- Motivation
  - Skew in component sizes can yield unnecessarily high communication cost in component detection algorithms
  - Graphs with skewed component sizes typically exhibit a giant component containing a significant fraction of the vertices in the graph
- Optimization
  - Designed to detect giant components efficiently by starting computation from a single vertex

Optimization: Single Pivot

- Implementation
  - Picks a single vertex (pivot) randomly and finds the component that pivot belongs to by propagating its Id along its neighbors
  - Once component of pivot is found, original algorithm is used for remainder of the graph
- Weakly Connected Components, Strongly Connected Components, and similar algorithms can be optimized using this technique

Optimization: Single Pivot

- Cost Analysis
  - If pivot vertex is picked from the giant component, all unnecessary propagation messages and computation costs for detecting the giant component are avoided
  - If pivot is not picked from the giant component, parallelism of the algorithm is decreased as instead of multiple components, just pivot's component was detected in the iteration

Experiments

- Experimental Setup
  - Three clusters
    - Large-EC2 (four virtual cores, 7.5GB RAM)
    - Medium-EC2 (two virtual cores, 3.75GB RAM)
    - Local (32 cores, 64GB RAM)
  - OS: Red Hat Linux
  - Fault Tolerance: Off
  - Graph Partitioning: Random

Experiments

- Strongly Connected Components
  - Finishing Computations Serially (FCS): 1.3x to 2.3x runtime reduction. 28% to 56% supersteps reduction on web graphs
  - Single Pivot (SP): 1.1x to 2.1x runtime reduction
  - FCS + SP: 1.45x to 3.7x runtime reduction

Experiments

- Minimum Spanning Forest
  - Storing Edges at Subvertices (SEAS): 1.15x to 3x runtime reduction
  - SEAS + Edge Cleaning on Demand: 1.2x to 3.3 additional run-time benefit. Increase in communication cost by 1.03x

Experiments

- Graph Coloring
  - Finishing Computations Serially: 1.1x to 1.4x runtime reduction. 10% to 20% supersteps reduction

Experiments

- Approximate Maximum Weight Matching
  - Edge Cleaning on Demand: 1.45x runtime reduction. 1.3x to 3.1x communication cost reduction. 1.7x to 2.2x increase in number of supersteps

Experiments

- Weakly Connected Components
  - Single Pivot: 2.7x to 7.4x runtime reduction

Criticism

- Any arbitrary graph algorithm can be implemented using Pregel – no proof has been provided
- Master – single point of failure
- What if master fails – no clear details mentioned
- Maximum Weight Matching – Finishing Computations Serially can be applied – Authors also comment the same, yet results are not shown for this algorithms

Conclusion

- Vertex-centric computation model – Think like a vertex
- Message passing between vertices
- Fault tolerance mechanism
- Optimization techniques
  - Communication cost
  - Number of supersteps

Questions