



Pregel: A System for Large Scale Graph Processing

Presented by
Karthik Mohan



Real World Graph Processing

- **Web Graph:** PageRank
- **Social Graph:** Popularity rank, Personalized rank, Shortest paths, Shared connections, clustering and Propagation.
- **Advertisement:** Target Ads
- **Communication Network:** Maximum flow, Transportation routes
- **Biology Network:** Protein Interactions
- **Pathology Network:** Finding Anomalies



Challenges in Graph Processing

- Poor locality of memory access.
- Very little work done per vertex.
- Changing degree of parallelism over the course of execution.



Existing options

Options	Cons
Custom Distributed Infrastructure	Requires implementation effort for new Algo. / new graph representation
Existing Distributed Computing platform <ul style="list-style-type: none">• Map Reduce -> Suboptimal performance and Usability issues.	Not suitable for Graph processing.
Single Computer Library	Scalability issue.
Existing Parallel Graph System	Addresses parallel graph algorithm, but not fault tolerance and other issues.



Solution: Pregel

- Developed at Google
- Inspired from Bulk Synchronous Parallel (BSP) computing model
- Distributed message passing system.
- Vertex-centric computation. Vertex are the first class citizens.
- Scalable and fault tolerant.




Bulk Synchronous Parallel

- Computations consist of a sequence of iterations, called Superstep.
- During a superstep, framework calls user-defined computation function on every vertex.
- Computation Function specifies behaviour at a single vertex V and a single superstep S .
- Supersteps end by Barrier Synchronization,
- All communications are from S to $S+1$



Pregel Computation Model

- Computation on locally stored data.
- Computations are in-memory.
- Terminates when all the vertices are inactive or no messages are in transit.
- The Input Graph representation is partitioned by the master and distributed among the workers using $\text{hash}(\text{ID}) \bmod N$, where N is the number of partitions (default partitioning)

- 
- Barrier Synchronization
 - Wait and synchronize before the end of a superstep.
 - Fast processors can be delayed by slower processors.
 - Persistent data is stored in GFS/BigTable.
 - Temporary data is stored in Disk.

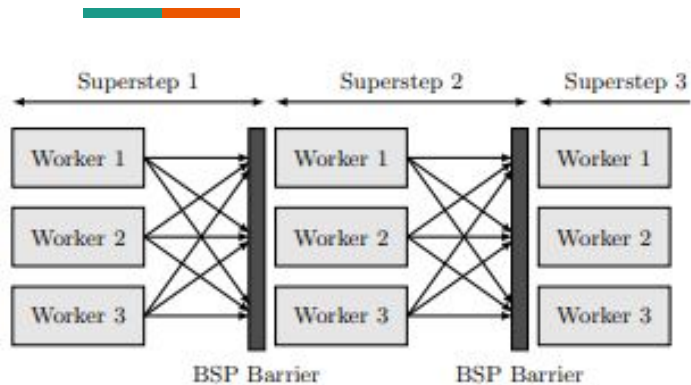
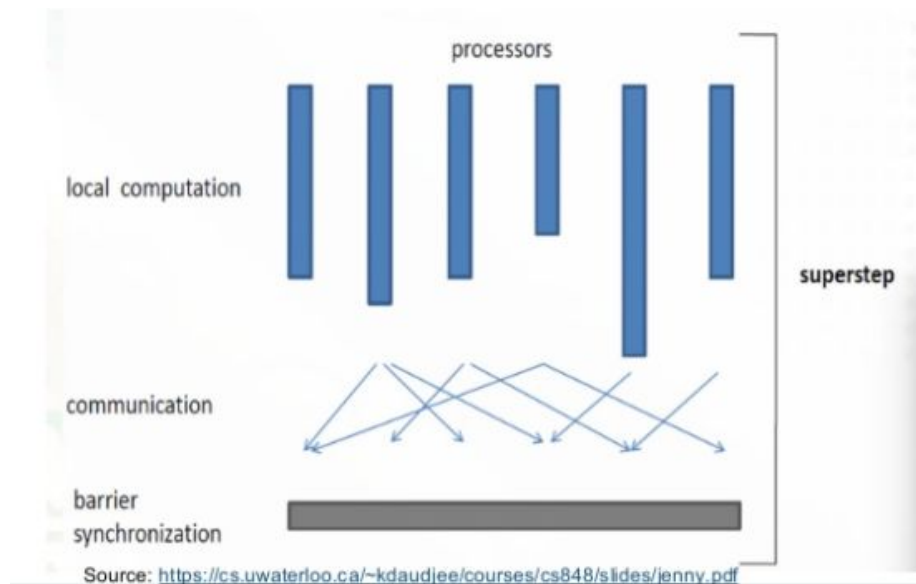


Figure 1: Basic computation model of Pregel, illustrated with three supersteps and three workers [23].

Source: "An Experimental Comparison of Pregel-like Graph Processing Systems*"

<https://www.vldb.org/pvldb/vol7/p1047-han.pdf>



Representation of a One Superstep



Pregel's C++ API

- Subclass Vertex class.
 - Template arguments: Vertices, Edges, Messages.
- Users override compute(), executed at each active vertex in every Superstep.
- Predefined vertex methods allow compute() to query info about current vertex value and it's edges, and also send messages to other vertex.

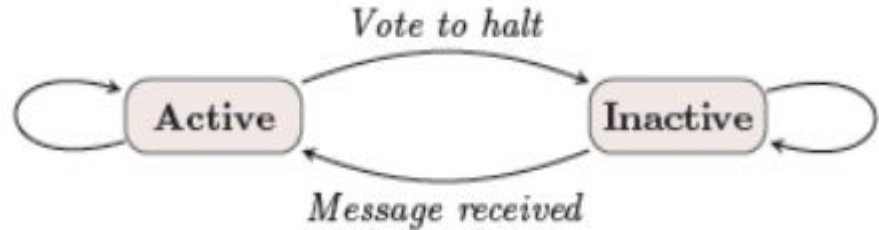


Properties of a Vertex

- Can mutate local value and value on outgoing edges.
- Can send arbitrary number of messages to any other vertices.
- Receive messages from previous superstep.
- Can mutate local graph topology.
- All active vertices participate in the computation in a Superstep.

Vertex State Machine

- Initially, all vertices are active.
- A vertex can deactivate itself by voting to halt.
- Deactivated vertices don't participate in the computation.
- Vertices are reactivated upon receiving a message.





Messages

- Consists of a message value and destination vertex.
 - If the dest. Vertex is missing, user defined handlers are used.
- Typically sent along outgoing edges.
- Can be sent to any vertex whose identifier is known.
- Are only available to receiver at the beginning of the superstep.
- Guaranteed to be delivered at most once.
- Can be out of order.



Combiner

- Sending messages incur overhead.
- System calls `Combine()` function for several messages intended for a Vertex `V` into a single message which is then transmitted.
- No guarantee which group of messages will be combined or the order of combination.
- Should be used for commutative and associative messages.
- Not enabled by default. Custom implementation is possible.



Aggregator

- Mechanism for global communication, monitoring and data collection.
- Used for statistics. Ex: Total number of edges, generate histogram of statistics.
- Provides Min, Max and Sum aggregators by default for Int and Str types.
- Aggregators can be used for global coordination. For Instance, Compute() until Aggregator determines all V's satisfy some condition, followed by a different compute()
- By default, single superstep, but sticky aggregators are also possible. Ex: Mutated edge count




Topology Mutations

- Vertices can dynamically create/destroy vertices, edges.
- Mutations and conflict resolution take place at barrier.
- Except local mutation immediately takes place.

Order of Mutations:


- Edge deletion
- Vertex deletion
- Vertex addition
- Edge addition

- 
- Multiple vertices can issue conflicting requests in same superstep. Ex: Two requests to add V with different Initial values.
 - Determinism achieved using partial ordering and handlers.
 - For the above case, system picks one arbitrarily.
 - Coordination mechanism is Lazy.
 - Global mutation: Mutation of other vertices.
 - Local mutation: Vertex mutating itself.
 - Global mutations do not require coordination, until they are applied.



Master Implementation

- Partitions the input graph and assigns one or more partitions to each worker.
- Keeps track of:
 - All alive workers
 - Worker's unique identifiers
 - Addressing Information
 - Partition of the graph that is assigned to the worker.
- Coordinates barrier synchronization.

- 
- Fault tolerance by checkpoint, failure detection and reassignment.
 - Maintains statistics of the progress of computation and the state of the graph.
 - Doesn't participate in computation.
 - Not responsible for load-balancing.



Worker Implementation

- Responsible for computation of assigned vertices. Maps vertex ID's to State
- State of a vertex: Current value, List of outgoing edges, Incoming Msg Queue, Flag
- Keeps two copies of active vertices and incoming messages. One for the current superstep and one for the next superstep.
- Places local messages immediately in the incoming message queue.
- Buffer remote messages and flush messages asynchronously if a threshold is reached.



Normal Execution: (Under absence of faults)

- Many copies of user program is made on a cluster of machines. One of them is master. Workers use cluster management system's name service to discover master's location and send messages to master.
- Master determines the partitions for the graph, and assign one or more partition to each worker.
 - More than one partition allows parallelism among partition and better load balancing.
- Upon Instruction from master, Workers perform superstep and executes `compute()` on V's and manage messages.
- When finished, reports the number of vertices active in next time step to the master.
- Upon termination, master instructs each worker to save it's portion of graph.




Fault Tolerance

- Checkpoint at the beginning of the superstep.
 - Master saves aggregator values separately.
 - Worker saves vertices, edges and incoming messages.
- Worker failure is detected by ping messages.
- Recovery
 - Master reassigns failed worker partition to other available workers.
 - All workers restart from Superstep S by loading state from the most recently available checkpoint.



Confined Recovery:

- Recovery is only confined to lost partitions.
- Workers also save outgoing messages.
- Recomputes using logged messages from healthy partitions and recalculated ones from recovering partitions.



Applications: PageRank

Message Type: double

Vertex value Type: double

Edge Value Type: void

Executes for 30 supersteps.

Initial values at **S0**: $1/\text{No. Of vertices}()$
From **S1 to S30**: $0.15/\text{No. Of vertices}() + 0.85 * (\text{Sum of values of Incoming messages})$

```
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();
    }
}
};
```

Figure 4: PageRank implemented in Pregel.



Single Source Shortest Path

S0: Source vertex value INF -> Zero and send updates to its neighbors.

In the following supersteps, if the current vertex value > (min. Of Incoming messages) update the vertex value and pass on the potential updates with weight of each outgoing edge added to the new minimum distance.

Terminate when no more updates occur. Guaranteed to terminate if all edge weights are non negative.

```
class ShortestPathVertex
: public Vertex<int, int, int> {
void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
        mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
        *MutableValue() = mindist;
        OutEdgeIterator iter = GetOutEdgeIterator();
        for (; !iter.Done(); iter.Next())
            SendMessageTo(iter.Target(),
                           mindist + iter.GetValue());
    }
    VoteToHalt();
}
};
```

Figure 5: Single-source shortest paths.

```
class MinIntCombiner : public Combiner<int> {
virtual void Combine(MessageIterator* msgs) {
    int mindist = INF;
    for (; !msgs->Done(); msgs->Next())
        mindist = min(mindist, msgs->Value());
    Output("combined_source", mindist);
}
};
```

Figure 6: Combiner that takes minimum of message values.



Maximal Bipartite Matching

Match two distinct set of vertices with edges only between the sets, and output the edges with no common endpoints.

Maximal: No additional edge can be added without sharing an endpoint.

Message Type: boolean

Vertex value Type: Tuple (L/R, Name of matched V)

Edge Value Type: Void (carry no information)

Phase 0: L's sends messages to its neighbors requesting (send messages) to be matched. If it's already matched and has no outgoing edge, sends no message.

Phase 1: Each available R randomly chooses one of the requests, sends a grant message to that requestor and denying to others.

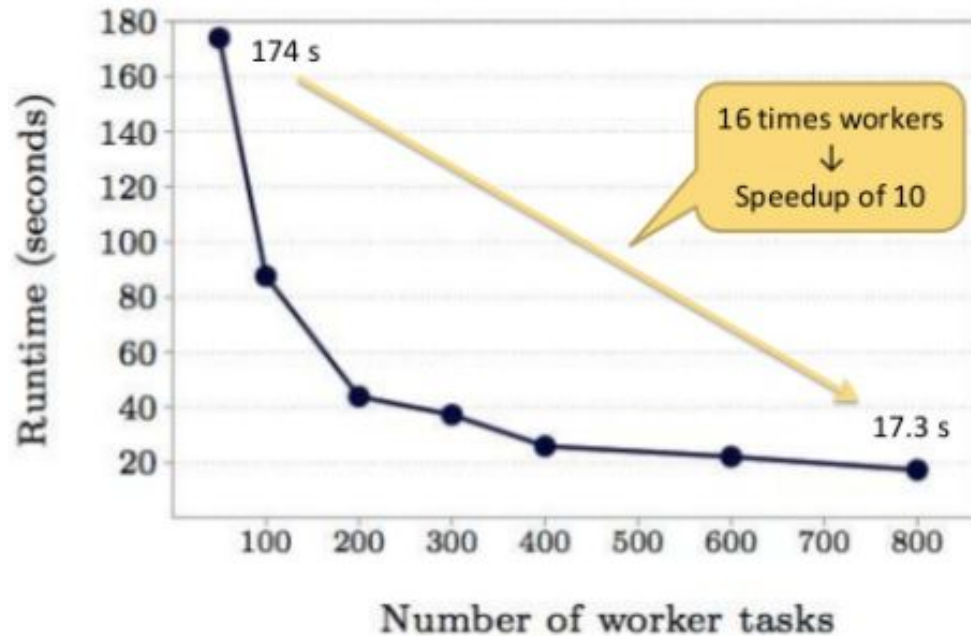
Phase 2: L's chooses one of the grant it receives and sends an acceptance message. L's that don't send messages in phase 0, do not enter this phase.

Phase 3: Unmatched R receives at most one acceptance message, notes it and halts.

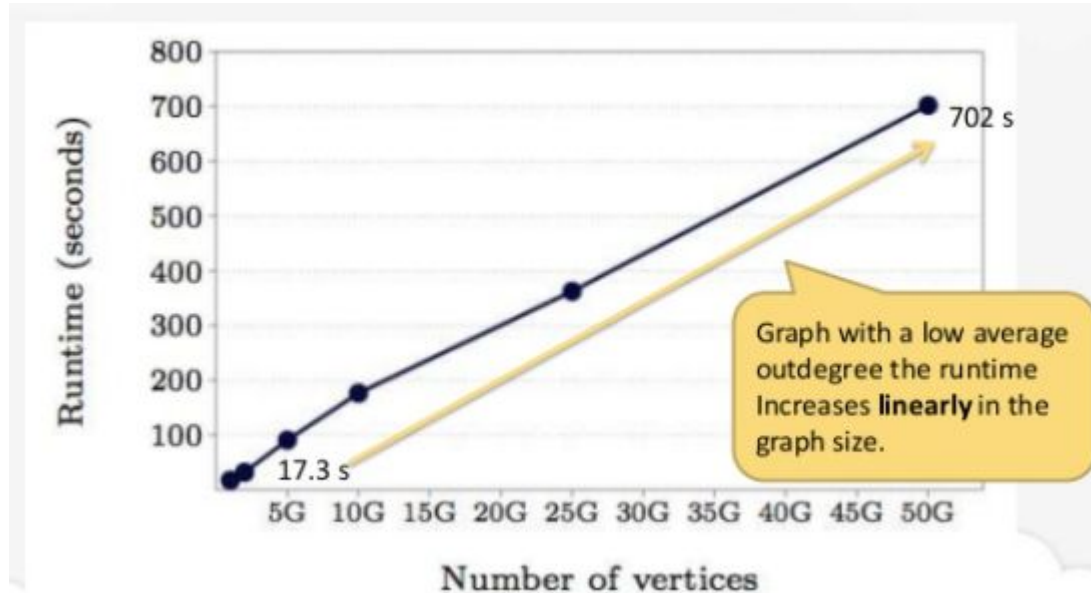
Performance

- Experimental Setup:
- Hardware: A cluster of 300 multi-core commodity PCs.
- Algorithm: SSSP with unit edge weights.
 - All-pairs shortest paths impractical for large graphs, because of $O(|V|^2)$ storage.
- Measures scalability with respect to both the number of workers and number of vertices.
- Data Collection:
 - Binary Trees - To test scalability
 - Log-normal random graphs - To study performance in realistic setting.
- No checkpointing.

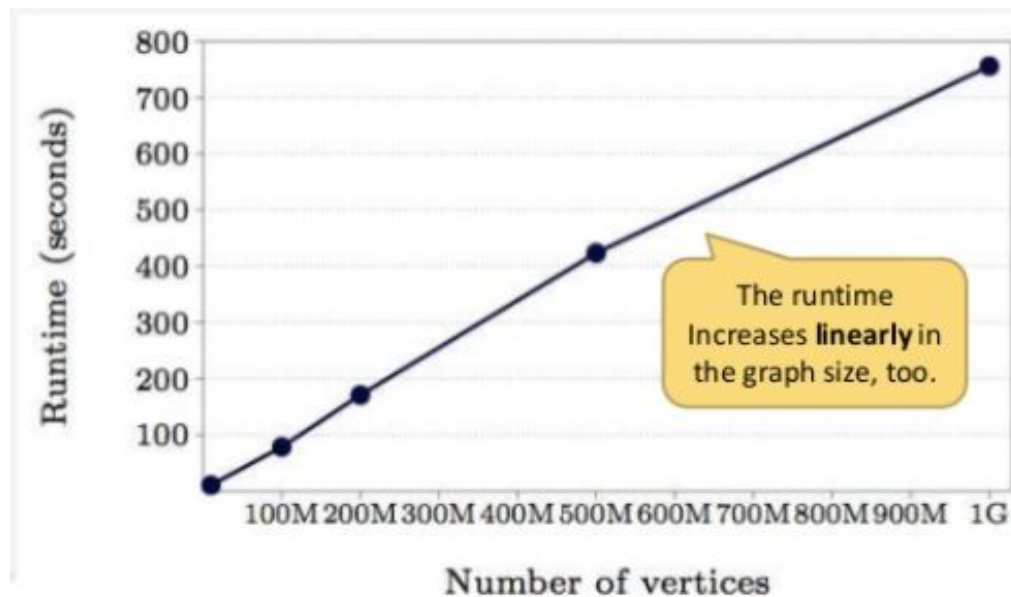
SSSP - 1 Billion Vertex Binary Tree



SSSP - Binary Trees: Varying graph size of 800 worker tasks



SSSP - Log Normal Random Graphs: Varying graph sizes on 800 worker tasks (Mean outdegree = 127.1)





Summary

- Distributed System for large scale graph processing.
- Vertex centric BSP model
 - Message passing API
 - A sequence of supersteps
 - Barrier Synchronization
- Coarse grained parallelism
- Fault Tolerance by checkpointing.
- Runtime performance scales near linearly to the size of the graph. (CPU bound)



Discussion:

Question 1:

What is the key to Pregel's scalability?



Answer 1:

The key to the scalability is batch messaging. The message passing model allows Pregel to amortize latency by delivering messages asynchronously in batches between supersteps.




Question 2:

Does pregel work well with Sparse or Dense graphs?



Answer 2:

- Dense graphs can be considered potentially a difficult setting for a pregel computation, assuming the algorithm tends to send messages over most vertices and their edges at each iteration.
- Given the synchronous nature of the computation, all messages produced during the previous iteration need to be produced and stored before they can be consumed at the following iteration. On massive datasets where resources are pushed to the limit, this can be a problem.
- In a distributed setting, this can mean the computation can be bound on network IO, and a pregel computation is already mostly network bound.
- These aspects can be reduced by a good partitioning of the data that exploits locality, hence reducing the number of messages that hits the network and maximising message sharing within a worker, but good graph partitioning can be hard on realistic graphs.

- 
- Finally, the problem of message staging can be reduced by relaxing the synchronicity constraints, and it does require to break some assumptions on the programming/computation model.
 - Pregel shines on sparse graphs that tend to have small world and scale free properties (like a social network or the www) where average degree and diameter are sensibly low, and locality tends to be higher.



Question 3:

Does Pregel guarantee serializability, if so, is it efficient?



Answer 3:

No. Pregel does not provide serializability. Although, other frameworks like Giraphx (bypasses the message queues in BSP and reads directly from the worker's memory for the internal vertex executions) and other research following the Pregel paper (2010) improves on providing serializability.

Source: Providing Serializability for Pregel-like Graph Processing Systems



Question 4:

What are some potential attributes of algorithms that would prevent Pregel from being efficient in modelling them? ("Think as vertex")



Answer 4:

- All these systems divide input graphs into partitions and employ a “think like a vertex” programming model to support iterative graph computation. This vertex-centric model is easy to program and has been proved useful for many graph algorithms. However, this model hides the partitioning information from the users, thus prevents many algorithm-specific optimizations. This often results in longer execution time due to excessive network messages. (e.g., in Pregel)
- It does not always perform efficiently, because it ignores the vital information about graph partitions. Each graph partition essentially represents a proper subgraph of the original input graph, instead of a collection of unrelated vertices. In the vertex-centric model, a vertex is very short sighted: it only has information about its immediate neighbours; therefore, information is propagated through graphs slowly, one hop at a time. As a result, it takes many computation steps to propagate a piece of information from a source to a destination, even if both appear in the same graph partition.
- Source: From "Think Like a Vertex" to "Think Like a Graph"
<https://researcher.watson.ibm.com/researcher/files/us-ytian/giraph++.pdf>



Question 5:

What are the important issues that are not clearly addressed in this paper?



Answer 5:

- Master Failure (Paxos? Replication?)
- If fault tolerance occurs, it is not clear whether only the work for the reassigned graph partition, or the entire work for that superstep is recomputed.
- Does not address when infinite loops might occur and how to account for them.



Question 6:

Is Pregel really programmer friendly?



Answer 6:

A major downside for Pregel is that it offloads a lot of responsibility to the programmer. The programmer must develop code for this decentralized vertex-mode with round-based messaging. This model leads to some race-conditions (mutating two vertices in the same superstep) and those conflicts are also left to the programmer to deal with handlers.