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

Google, Inc.
2010

What is Pregel?

- A System for Large-Scale Graph Processing.
- An iterative and graph specific version of MapReduce.
- A distributed implementation of the Bulk Synchronous Parallel model (BSP).
- Efficient, scalable and fault-tolerant.

Graph Examples

- Web Graphs.
- Social networks.
- Transport networks.
- Similarity of newspaper articles.
- Paths of disease outbreaks (epidemiology)
- Citation relationships.

Algorithms

- Maximum Value.
- Shortest Path.
- Clustering.
- Variations of Page Rank.
- Minimum Cut.
- Connected Components.

Graph processing challenges

- Poor locality of memory access.
- Low compute to communication ratio.
- Changing degree of parallelism over the course of execution.

Previous Options

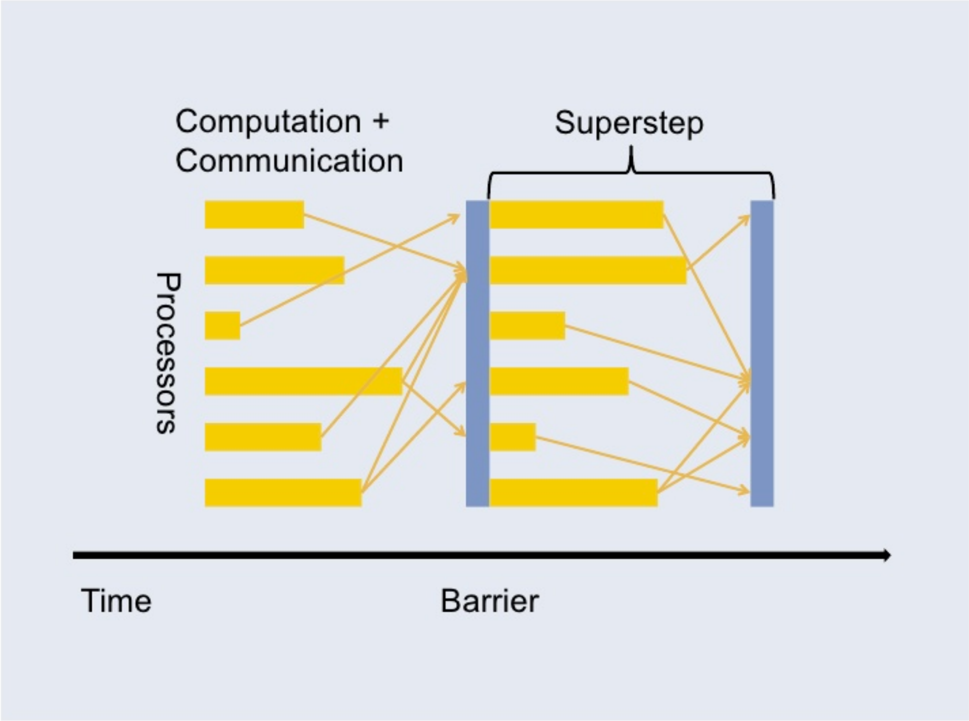
- Craft a custom distributed infrastructure.
  - Lots of effort.
  - Have to re-implement for each new algorithm or graph representation.
- Use existing distributed computing platform such as MapReduce.
  - Can lead to sub-optimal performance and usability issues.
  - Better fit would be a message passing model.
- Use graph algorithm libraries for use on a single machine.
  - Severely limits scale.
- Use existing parallel graph system.
  - No fault tolerance or support for other distributed system problems.

Pregel's solution

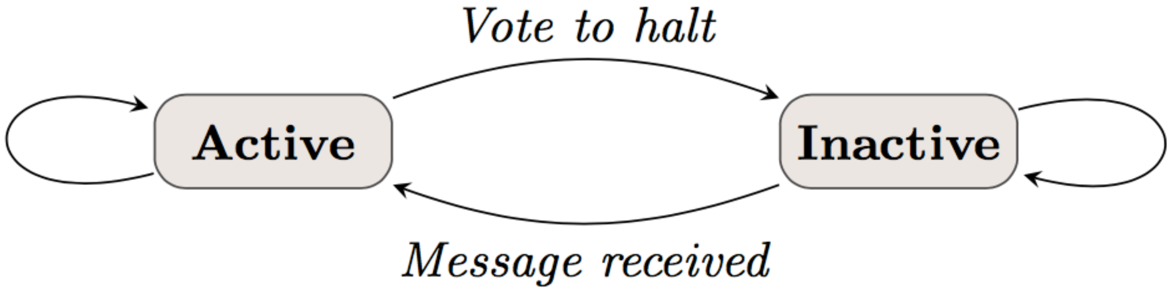
- Implement a scalable and fault-tolerant platform with an API that is sufficiently flexible to express arbitrary graph algorithms.
- Just like MapReduce, take care of all distributed problems behind the scenes.
- Present simple functions to be filled in by the programmer.
- Designed to be optimal for graphs.

Pregel Computation

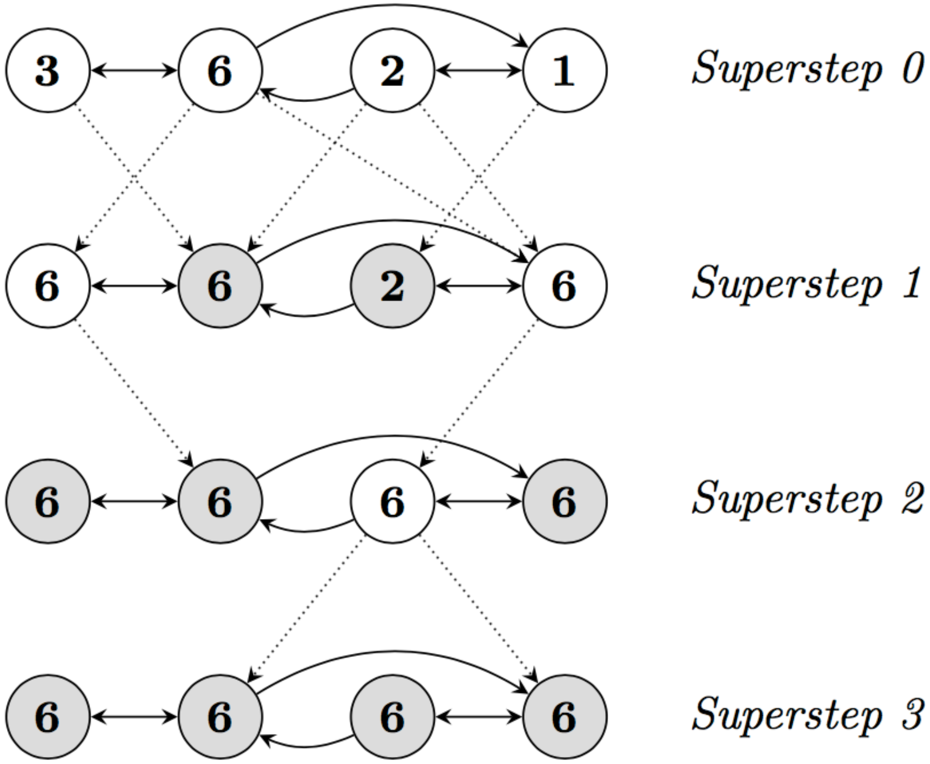
- One Master <-> Many workers.
- Master synchronizes workers, each worker performing a computation in each **Superstep**.
- Worker's send messages between themselves.
- Iterates until all vertices vote to halt a and there are no messages in transit.



Vertices



- Has a modifiable value and a list of its outgoing edges and their values.
- Only computes when active.
- All perform the same function.
  - Receives all messages sent to it in the previous superstep.
  - Performs computation.
  - Sends messages (most likely along outgoing edges).
  - Optionally vote to halt.
- Can request to add/remove vertices/edges.



Example: PageRank

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

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

Other Aspects

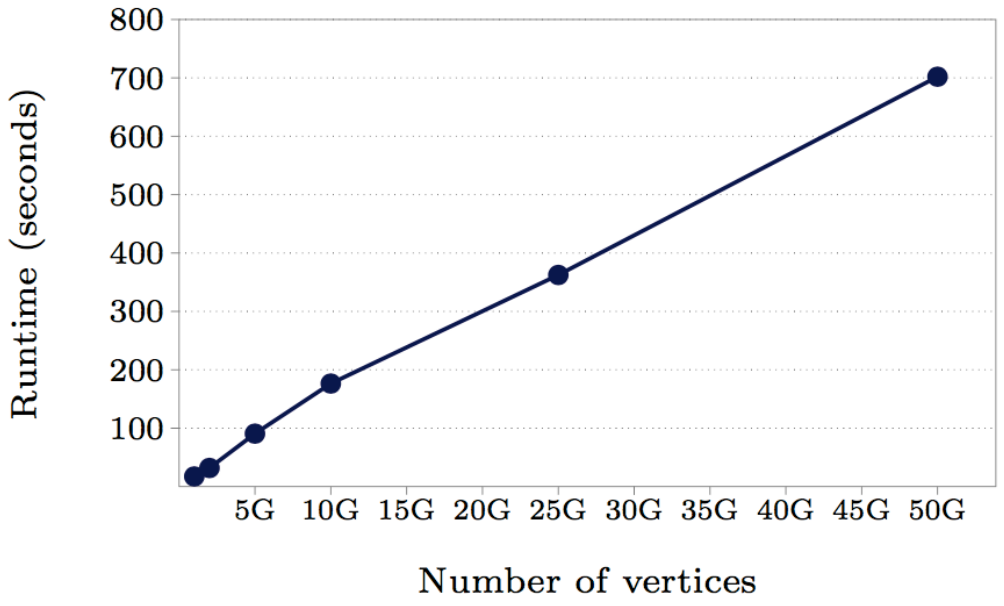
- Message Passing
  - Delivered in asynchronous batches using buffer .
  - No order guarantees.
- Combiners
  - Combines messages headed for destination.
  - No guarantee it will happen.
- Aggregators
  - Master can aggregate data passed to it by workers.
  - Statistics, coordination, leader assignment.
- Status Page

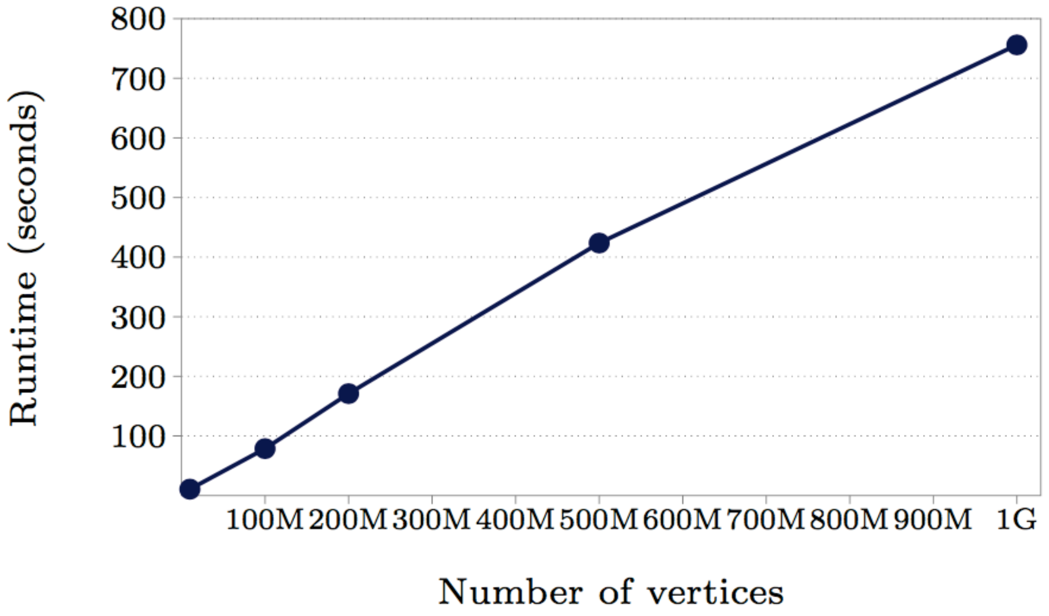
Other Aspects

- Graph Partitioning
  - Uses default hash on ID.
  - Can be replaced to get better locality.
- Fault tolerance
  - Check-pointing to persistent storage.
  - Failures detected using pings.
  - Frequency automatically calculated by mean time to failure model.
  - Confined recovery being looked into.

Performance

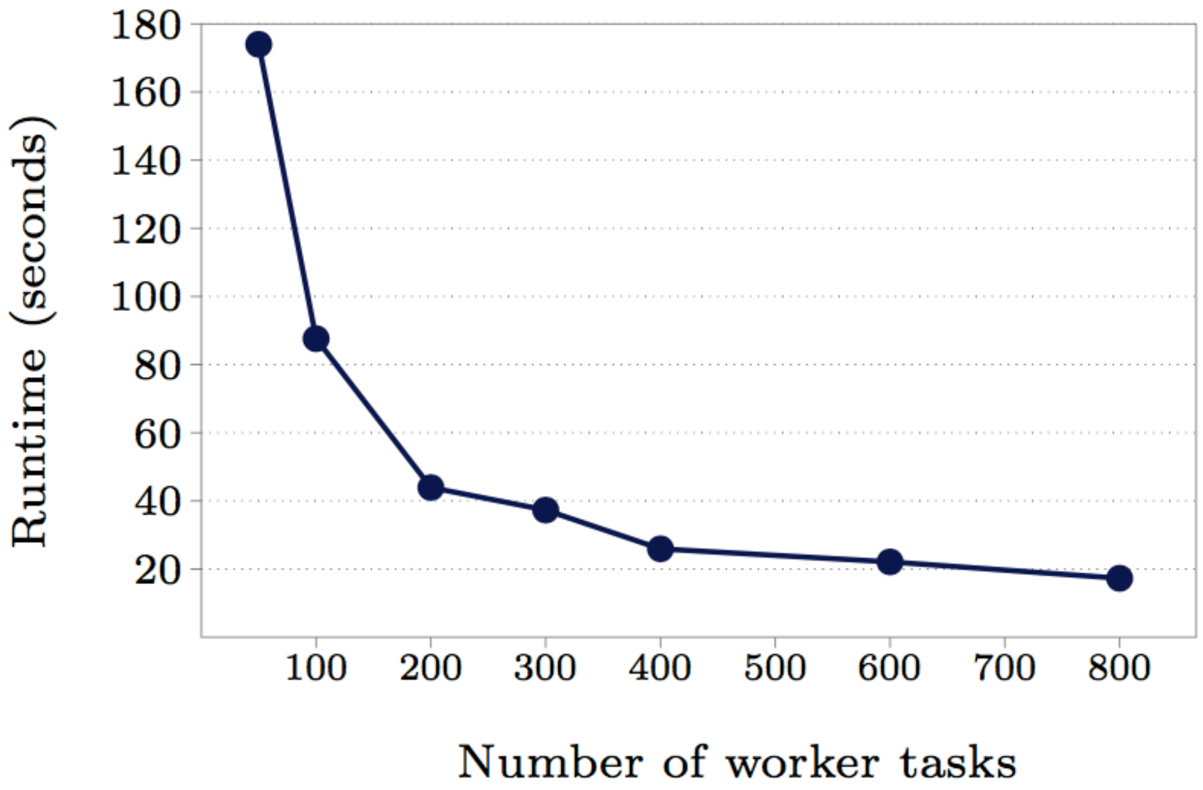
- Tested using Single Source Shortest Path Algorithm and default partitioning hash.
- Using binary tree and log-normal random graphs.
- Gives linear runtime increase for increasing graph size for both.
- Gives poorer performance for denser graphs.





Performance

- For binary tree on fixed number of machines.



Criticism

- Master is a single point of failure.
- A lot of network communication, especially for dense graphs.
- Still more limited (less expressive) than systems created later.
- Hard to partition the graph in a way that takes advantage of locality.
- Synchronicity slows all workers to the slowest worker.
- No way to redistribute load between workers.
- Performance not tested against any other systems or implementations.

Questions?