# BAG:EL

# Best Algorithms for Graphs: Easy Learning

Benny Wong, Gokce Dilek, Meg Thibodeau, Ryan Kim, Shaunak Tulshibagwale

## Introduction

***Distributed Graph Processor***

Graphs can represent many different types of data, from proteins to Facebook friends, and distributed graph processing is key to the core functionality of many large companies. Distributed graph storage and processing are increasingly important to handle large datasets. We built BAG:EL is a distributed graph processor that is modelled on the Pregel library. BAG:EL uses a vertex-based implementation of the Bulk Synchronous Parallel model to solve iterative graph problems. In order to focus on the distributed aspects of the project, we limited queries to calculate the shortest path between nodes and the PageRank of specific nodes, as described in the Pregel paper[1].

## Design

The structure and organization are based on the Pregel API, a graph-processing system designed by Google for handling large-scale graph problems using a distributed network. The system will contain three types of nodes: multiple worker nodes, a single coordinator node, and a single client node.

In BAG:EL, clients can send a shortest path or PageRank query to a central coordinator node. The coordinator node handles connections to up to $2^{32}$ worker nodes and synchronizes the computation through a series of supersteps before returning the query result to the client. Workers are responsible for a subset of vertices which are assigned through a hash at the start of each query. During a single superstep, workers execute an iteration of the algorithm specific to the query type on all assigned vertices and send messages to other parts of the graph in a “superstep”. The succeeding superstep uses the results of the work and messages passed to vertices to set up and complete the next iteration of the algorithm for each vertex. Using supersteps synchronizes the process across vertices to avoid timing issues.

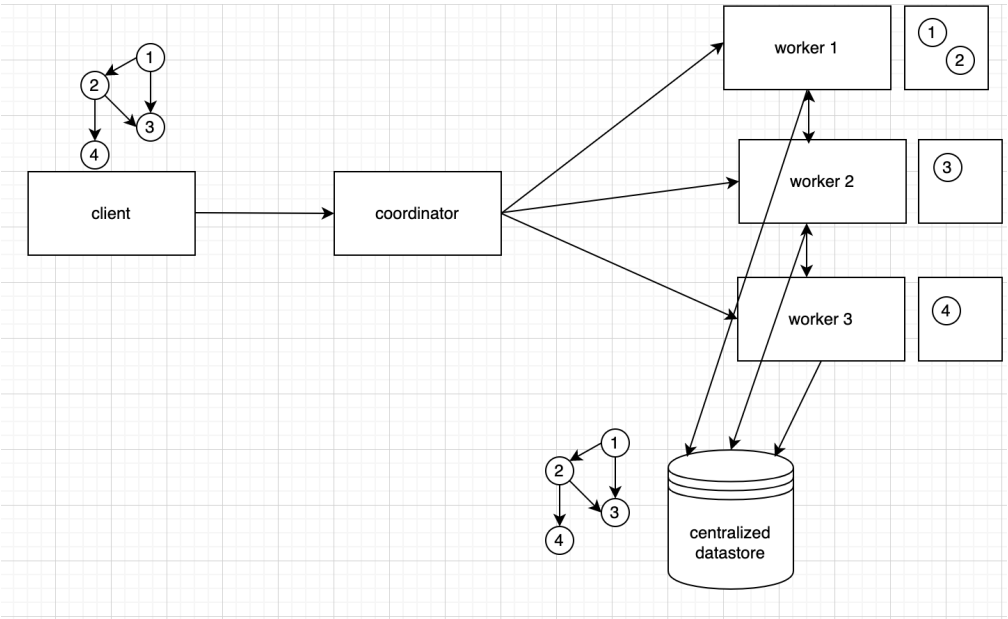
**Assumptions:**

- The coordinator node does not fail
- Worker processes can fail
- Worker nodes (the machines) cannot fail
- The number of workers participating in a client query is known at the beginning of the computation and is fixed for the duration of the computation.
    - Worker processes that fail will come back online to resume the computation

    - New workers can join during the current computation but will not be used until the next query
- Network communication does not fail (no network partitions)
- The graph that the client wishes to perform computations on has already been parsed and stored in the centralized database
- The number of workers and the lengths of paths fit inside an int32 variable
- The client does not fail and will wait indefinitely to receive a response from the coordinator node to the queries
- There will be at least 1 worker node available during a superstep

# Implementation

### Architecture Overview




### Coordinator (Master) Node

One of the nodes in the system is a dedicated coordinator node which tracks the worker nodes, which are the majority of the other nodes in the system. It is assumed that the coordinator node will not fail. The primary functionalities of the coordinator node are as follows:

1. *Centralized state management:* The coordinator stores the list of worker nodes that are currently known to be alive, in the form of a list of worker IDs and network addresses
2. *Fault tolerance:* The coordinator monitors the health of the worker nodes using a variant of the *fcheck* library we developed in the previous assignments. If a node is detected as failed, the coordinator will instruct all active worker nodes to roll back to the last checkpoint that was saved in persistent storage by all the query workers. When reverting back to a previous superstep, the coordinator will block the computation from proceeding

until all of the worker processes are available again. Fault tolerance and handling are covered in more detail below.

3. *Initiate the computation:* When a client issues a request, the coordinator instructs the worker nodes to get the vertices from the centralized database and set up the worker state as well as the initial state for each of their assigned vertices. This initial state will depend on which query was requested by the client.
4. *Computation State:* The coordinator will also be responsible for handling the current state of computation based on messages from the worker nodes. At the end of each superstep, the message received from a worker includes the superstep number, a flag to indicate if the worker is active, and the current value of the computation at the vertex we wish to monitor (for Shortest Path, this is the destination vertex in the query, and for Page Rank, this is the only vertex in the query). The coordinator will save the value returned from the worker that has this target vertex as part of its partition.
5. *Proceed with the computation:* Once all workers return messages to the coordinator, this will mark the end of the current superstep. At this point, if no messages were generated by the previous superstep, the computation will reach an end and the coordinator will send the result back to the client. Otherwise, the coordinator will instruct the workers to execute the next superstep for active vertices and vertices receiving messages.

## Worker Nodes

### Worker Join Protocol

To join, a worker will start an fcheck node at the configured local address, and send a join request to the coordinator over RPC. The join request contains the worker ID and addresses used to communicate. The coordinator will wait until the current query is completed by the currently active workers before it will allow new workers to join. We also extended the worker join process to account for worker process failures that might happen when a client query is in progress. The coordinator keeps track of two sets of worker nodes; one set consists of all worker nodes known to the coordinator, and one set consists of all worker nodes that are involved in the current client query (since we assume a fixed number of workers during a client query). When the coordinator receives a join request, if the joining worker is a brand new worker, it is simply added to the first set of workers without affecting the progress of the ongoing query. If the joining worker is a previously failed worker process that is part of the current query, the coordinator (re)adds the worker to both of the sets, and invokes a procedure to reset the worker state to the last known checkpoint state.

### Worker Normal Operation

The worker ID is used to partition the set vertices that each worker is responsible for. A worker is responsible for a vertex if *hash(VertexID) mod N == WorkerID.* The worker IDs will start at 0 and increase to N-1, inclusive. The worker stores a map from vertex ID to the state of each vertex, where the state of each vertex consists of:

- the current value of the vertex
- a list of its outgoing edges
- a queue containing incoming messages (from incoming edges)
- a cache of the most recent incoming message from each in-neighbour of the vertex

- a flag specifying whether the vertex is active or inactive

*Note:* Two copies of the active vertex flags and the incoming message queue need to be stored, one for the current superstep and one for the next superstep.

When the coordinator node initiates the succeeding superstep:

1. Workers loop through assigned vertices and call the compute method for each active vertex or vertex that is the recipient of an incoming message.
2. The compute method takes in the vertex, any incoming messages, and a way to send outgoing messages and executes a goal-specific algorithm (Page Rank, Shortest Path). This involves:
    a. Incrementing the superstep and sorting the incoming messages by vertex
    b. Updating the current value of a vertex based on incoming messages and cached previous messages
        - Shortest Path: if the lowest incoming value is less than the current value, update the current value
        - PageRank: Update the cache of all previously received messages, and sum over these messages to generate the new PageRank
    c. Send messages to all outgoing edges to be processed in the next superstep. Messages can also be sent to any vertex in the graph, regardless of distance.
        - Shortest Path: send *newValue+1* to each outgoing edge
        - PageRank: send *newValue/numOutNeighbors* to each outgoing neighbor
    d. Vertices with no messages sent are set to inactive
3. After the compute method has been executed for each vertex, the worker will update the coordinator with a message that includes the superstep number, a flag to indicate if the worker is active, and the current value of the computation at the vertex we wish to monitor, if that target vertex is located on the worker.
    a. The worker is active if it has sent messages and the superstep is less than the MAX_ITERATIONS defined in the code. The MAX_ITERATIONS constant could be easily removed to allow for more precise calculations, but it is in the deliverable code to ensure that the computation completes within the time allotted for the demo.
4. The worker waits for the coordinator to decide whether to continue with another superstep or start a new query.

## Clients

Clients are provided with the address of the coordinating node. Only one client request will be handled at a time. Users start the client and initiate a query using a single input on the commandline: ./bin/client [shortestpath|pagerank] [vertexId] [vertexId].

- Example commands:
    - ./bin/client pagerank 11
    - ./bin/client shortestpath 11 54

The result of the query is logged to the console and logfile.

### Fault Tolerance

Our assumptions in this project include that the coordinator and the client do not fail, thus, we are only concerned with analyzing worker failures.

Fault tolerance will be achieved through a process called *checkpointing*. Every n supersteps, the coordinator node will instruct the worker nodes to save the current state of their partition (state of their vertices) to their local storage.

*Failure Detection:* To detect failures, the coordinator monitors all worker nodes using a version of the fcheck library we developed in previous assignments. fcheck has been modified to allow the coordinator node to monitor more than one node at a time and will assign a random port for monitoring a server if no port was assigned in the Start struct.

*Failure Handling - Coordinator:* When the coordinator detects a failed worker node, it informs all workers to revert to the previous checkpoint. This allows surviving workers to revert while waiting for the failed worker which is more efficient, especially accessing checkpoint data from the database. The coordinator node will wait to continue computation until it receives a ready message from all workers, including the failed worker. On receipt of a ready message from all workers, it continues computation.

*Failure Handling - Worker:* When the coordinator node calls RevertToLastCheckpoint RPC for a worker, the worker retrieves the most recent checkpoint from the database, updates the state of its assigned vertices to the match the previous checkpoint, and reply with a ready message to the coordinator node.

### Datasets

For the underlying graph we will use Google's web graph dataset from 2002. The dataset contains 875,713 vertices and 5,105,039 edges (file size = 75.4 MB). The only data used from the dataset will be the vertexID and list of outgoing edges for each vertex.

### Hosting

Coordinator and worker nodes are hosted on different ugrad servers. The centralized database storing the graph is on Azure.

## Evaluation

The primary evaluation goal was ensuring that the system produced the correct results for a given query, and results were consistent and accurate in all conditions.

We utilized the testing package in the standard Go library to create unit tests for the computation results. Our unit tests check that the compute functionality provides the correct output for the given compute operation.

Evaluation of the BAG:EL system was done manually. As there is no GUI, BAG:EL includes logging of processes and results to a file (bagel.log) and to the console. The manual testing consists of starting the coordinator node followed by three or more worker nodes on different servers. We tested queries while adding new worker nodes to the system via the coordinator and removing nodes from the system by manually terminating the processes. Workers must be manually restarting using the standard ./bin/worker <id> command.

The table below contains the expected behavior for the component types on initialization, during a query, and after a failure. The correct result for a query should be produced and returned to the client in any of these cases, unless otherwise specified.

| **Coordinator Node** | **Worker Node** | **Client** |
|---|---|---|
| **Initialization:**<br>● Listens for worker and clients<br>● Can join 3+ worker nodes<br>● Waits for 1+ workers to join before starting a new query | **Initialization:**<br>● Start fcheck heartbeat response<br>● Join Coordinator<br>● Waits for Coordinator to start a new query | **Initialization:**<br>● Prints correct usage syntax if given invalid query<br>● Connects to Coordinator with valid query |
| **Query:**<br>● Validate vertex ids in query<br>● Notify all workers to start query<br>● Synchronizes supersteps by waiting for ready message from all workers in query before signalling to proceed to the next superstep<br>● Allows new workers to join during query but does not add them to computation until new query started | **Query:**<br>● Access vertices from centralized database using hash(VertexID) mod N == WorkerID<br>● Executes algorithm iteration on each vertex and passes messages to vertices as needed<br>● Messages Coordinator when finished iteration for each vertex and waits to proceed to next superstep | **Query:**<br>● Log result received from Coordinator node and end program<br>● If no path between given nodes exists, result is 2147483647 (maximum value of result type) |

To evaluate failure-handling, we looked at different cases.

**Worker Failure Handling Cases**

| Scenario | Coordinator Node | Worker Node |
|---|---|---|
| 1+ worker started, 1+worker fails while no query is ongoing | ● Log failure<br>● Wait for worker to rejoin before starting query | ● Failed worker rejoins as normal (through user manually starting worker process) |
| 3+ workers started, 1+ workers fail in order (a worker comes back up before another one fails) | ● Log failure<br>● Tell all workers to revert to most recent checkpoint and waits for all workers to be ready<br>● When failed worker(s) rejoins, Coordinator ensures that it sets up at the most recent checkpoint<br>● Restarts computation | ● Surviving workers access database and revert to most recent checkpoint<br>● Surviving workers message Coordinator when ready to begin computing from thmost recent checkpoint, then wait to proceed<br>● Failed worker rejoins (through user manually starting worker process)<br>● Failed worker loads checkpoint from database and messages coord when ready to continue<br>● Query computation continues until result produces |
| 3 workers started, [1-3] workers fail during a query one after another (a worker doesn't come back up before another one fails) | ● Log failure<br>● Tell all workers to revert to most recent checkpoint and waits for all workers to be ready<br>● Log second worker failure<br>● When failed worker(s) rejoins, Coordinator ensures that it sets up at the most recent checkpoint<br>● Wait for all failed worker to rejoin before starting query<br>● Restarts computation* | ● Surviving workers access database and revert to most recent checkpoint<br>● Surviving workers message Coordinator when ready to begin computing from the most recent checkpoint, then wait to proceed<br>● Failed worker rejoins (through user manually starting worker process)<br>● Failed worker loads checkpoint from database and messages coord when ready to continue<br>● Query computation continues until result produces* |

*indicates implementation of expected behaviour item is in progress (here there be a bug)

# Demo Plan

Logging in the console and in the file bagel.log is used to demonstrate the operation of the system by recording actions (ex: worker joining coordinator, coordinator detecting a failure, client sending query, client receiving query result). The config files used for components show the different servers used to host various components.

For each demo stage, we will demonstrate the main cases for queries:

- Shortest Path
    - Returns the shortest path length between given two vertices, or
    - Returns a value indicating no path exists between given vertices
- PageRank
    - Returns the page rank for the given vertex

**Normal Operation**

We will demonstrate how BAG:EL solves Shortest Path and PageRank queries with no worker failures. The coordinator node will be started, followed by 3+ worker nodes, and then a series of client queries will be sent.

**Failure Handling and Rejoining**

We will demonstrate how BAG:EL can handle 3 node failures between and during queries. Because the project made the assumption that worker failures will only be process-related and not machine-related, failed workers are always restarted. We will manually fail and restart 3 worker nodes while a query is not currently being computed. We will then fail 3 workers while a query is ongoing to demonstrate that the computation will halt until all workers rejoin, at which point the query computation will continue until workers produce a result.

**Joining**

The Failure Handling and Rejoining stages demonstrate how workers used in a current query can fail and rejoin. We will also demonstrate how BAG:EL can add additional workers during and between computing queries. Workers joining during a query will queue and only participate in computation once the query has finished and a new query has been received from a client. Workers joining while a query is not ongoing will participate when the next query starts.

# References

1. Pregel: A system for large-scale graph processing. https://www.dcs.bbk.ac.uk/~dell/teaching/cc/paper/sigmod10/p135-malewicz.pdf