Awesome Gamma

Heterogenous Dynamic Bulk Synchronous Parallel (BSP) programming in Go with a Pregel-inspired API

Mike Fink, Dorothy Ordogh, Graham St-Laurent, Bruno Vacherot

Introduction

Bulk Synchronous Parallelism (BSP) is a programming model used to distribute the execution of iterative parallel algorithms with minimal data races, and strong fault tolerance[2]. BSP is a foundational component of real world distributed systems, such as Google's Pregel[1] and Apache Giraph, which are both used to solve iterative graph problems. We are especially interested in Pregel's modelling of BSP.

At a basic level, Pregel functions by assigning vertices of a graph to a number of partitions, then assigning these partitions to multiple workers. The workers process their partition in parallel for one round of a "superstep". Each vertex can send messages to other vertices during the superstep. The superstep is completed when a synchronization barrier detects that all vertices have completed execution. At the beginning of the next superstep, the vertices have received all messages sent to them in the previous step and are allowed to proceed to the next superstep.

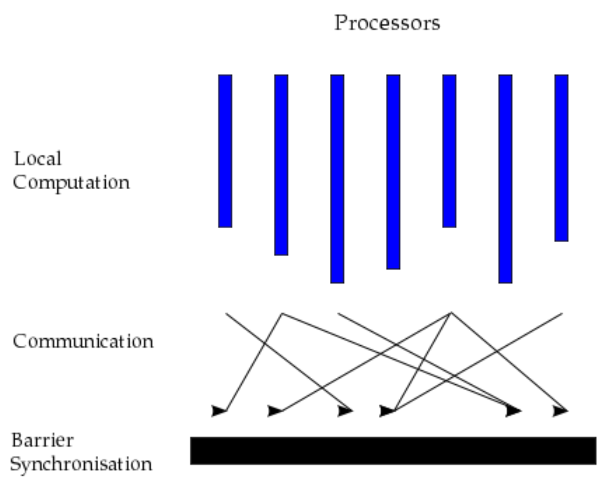


Illustration of a BSP superstep. Drawn by Discboy, specifically for the Bulk Synchronous Parallel Wikipedia page. https://en.wikipedia.org/wiki/File:Bsp.wiki.fig1.svg

We want to further our understanding of applicable distributed systems by creating a framework inspired by these systems. We will design and implement a Pregel-like Go API for distributed graph processing over heterogeneous resources. The API will allow users to create servers, workers, and clients of this service. A user can input a directed graph G to this service (and possibly some additional initialization data), which will return the result of running a graph algorithm on that input. Although our goal is to provide a service that could be used to run arbitrary iterative graph algorithms, we limit the scope of our project by only applying the framework to the PageRank[1] algorithm, in order to focus on the interesting distributed components of our system.

Heterogeneous Resources:
One of the distributed components we are focusing on is the use of heterogeneous workers - a faster machine should do more work. This is important for a system of this type because the barrier between supersteps limits a worker's ability to perform to its full potential if it has to wait for all other workers before moving forward. Workers cannot do additional work until the superstep is completed. Therefore we will need to design our server/superstep logic such it it distributes tasks unevenly among workers, as appropriate for their capabilities, or have a way for faster machines to request more work. We think this would be useful when the input size is much larger than the number of processes.

Worker Failures:
We want to intelligently deal with worker failures in the middle of our graph computation. If a worker fails then the computation for its assigned portion of the graph that had not been saved must be repeated. Pregel accomplishes this via a checkpoint-restart mechanism. Every N supersteps, workers save their portion of the graph to a global shared memory. When recovering, the portion of the graph that was not computed due to the failure is reassigned to active workers and all workers begin computation from the last checkpoint.

Additional distributed components we could explore if time permits:

Volunteer computing:
This system could potentially be used in a volunteer computing context, and ought to be somewhat resistant to byzantine worker failures. This can be addressed to some extent via replication of tasks. At the beginning of each superstep a worker would perform quorum on the messages it received and accept the majority result. This would essentially restrict non-deterministic algorithms from being used in the system because it would not be certain that a message from one vertex to another should be sent. The replication of tasks would likely be an alternative to checkpoint-restart because of the challenges inherent in having multiple nodes update a global shared memory. Alternatively, checkpointing could also be with a quorum of the replicated tasks, but performing this quorum would hinder performance since the graph data would have to be sent to a source to be compared.

Additionally, we would want to support worker nodes with intermittent connections. For example a node could drop connection in the middle of computation and once the connection is reestablished, be able to return its work.

Simultaneous clients:
Allow multiple clients to begin jobs with different graphs. The system should keep the computations separate and distribute the work amongst the workers so all jobs can run.

Project Overview

System Description

	Pregel	**Awesome Gamma**
Input	● Directed graph where each vertex is uniquely identified and is associated with a modifiable, user defined value ● The directed edges are associated with their source and target vertices, and a modifiable, user defined value	● Same
Superstep	● Computes vertex state in parallel and can modify vertex or edge state, receive messages sent in a previous step and send messages to be received in the subsequent step. ● Can modify graph structure by deleting or adding nodes	● Cannot modify graph structure/topology
Termination	● All vertices vote to halt. A vertex votes to halt if it has nothing further to do ● A vertex is reactivated if it receives a message	● Same
Output	● Set of values output by the vertices	● Same

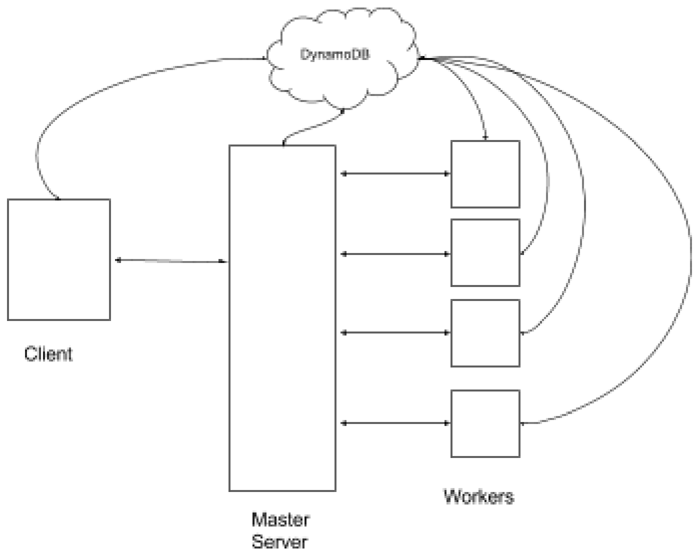
Implementation

	Pregel	**Awesome Gamma**
Architecture	● Google computer cluster, with distributed storage system for persistent storage ● Temporary data stored locally	● *Volunteer computers with a DynamoDB instance for persistent storage.* ● Temporary data stored locally.

Data Distribution	• Graph divided into N partitions and vertices assigned to partitions by `hash(vertexID) mod N` • One or more partitions assigned to each worker. Each worker is later assigned vertices to work on • Vertices are orthogonal to the contents of the partition	• Graph divided into N partitions and vertices assigned to partitions by `hash(vertexID) mod N` • *Partitions assigned to workers based on computational capability* • *Workers work only on vertices in their partition*
Participant Roles	• One cluster member is assigned to be the master • Workers discover and connect to the master • The master is not assigned any partitions, assigns partitions to workers and manages the process • Each worker maintains the state of its portion of the graph	• *A designated server is the master. Volunteers connect to the master* • The master is not assigned any partitions, assigns partitions to workers and manages the process • Each worker maintains the state of its portion of the graph
Superstep	• Workers call a Compute() function on each of its vertices • Messages for that vertex are received and the computation is performed • Messages are sent asynchronously directly to other workers • Tells master when superstep is completed	• Workers call a Compute() function on each of its vertices • Messages for that vertex are received and the computation is performed • *Messages are sent asynchronously to master, which then distributes them to the appropriate worker* • Tells master when superstep is completed
Post Superstep	• Master may instruct workers to save their portion of the graph	• Master may instruct workers to save their portion of the graph • *Master may redistribute vertices*
Failure Detection	• Ping messages are regularly sent from the master to each worker	• Broken TCP connection between Master and each Worker provides failure information
Fault Tolerance	• Checkpoint-restart	• Checkpoint-restart

Project Details

System Components
There are three main components to our system: clients, the master server, and worker nodes. Additionally, we will make use of a distributed database service such as AWS DynamoDB for our global persistent data storage, for example, for storing the graph for our PageRank service.



Client
The client connects to the server over TCP and sends it a graph -- likely in the form of a key in our global DB. Once the request is sent, the client waits for a the result from the server either via the global DB, or some sort of value. For example, in terms our PageRank example, the result could be a key to the global DB with an updated copy of the graph, where each vertex has its PageRank score. Only one client can be connected at a time.

Failures:

- If the client disconnects during our graph processing and reconnects to the server the server will provide the result when complete. If our system completes while the client is disconnected, it will cache the result for an indeterminate length of time.
- If a different client tries to reconnect after an interruption, but while the graph is being processed, it is rejected.

Extensions:

- Enable handling of multiple clients, working on multiple graph problems at a time.
- Returning cached results for identical client requests.

Master Server and Detailed Design

The master server accepts a connection from a client, receives a graph from the client and provisions it to the distributed database. It also receives connections from workers.

The server partitions the graph into N partitions, and assigns each partition to a worker. Since our model is an adaptation of BSP, the server initiates the supersteps. During a superstep each worker processes its assigned partitions. The server is responsible for relaying messages between workers, and keeping track of which worker has voted to halt. The algorithm completes when all workers vote to halt and there are no pending messages.

Failures:

- If the server fails, the system is down and the job fails. The workers discover this from their TCP connection and exit.
- In case a worker takes an exceptionally long time to complete, the server maintains a timeout for each superstep. If this worker's response time exceeds this timeout, then the server disconnects from it and handles it as described in the Worker Failures below.
- In case of worker disconnection, see Worker Failure below.

Extensions:

- Handle server restarts. Store checkpoint number and partition info in global DB. On server restart, wait for worker registration and compare against partition info to see if the workers have changed. Workers store all the messages from the last superstep so they can 'replay' them in the case of a server restart.
- Another option is to follow Pregel's example and elect Master Servers from the worker pool. This is more difficult for us, however, because we have a heterogenous worker pool, and we would need to determine eligible workers. Additionally, we would need to ensure that clients are still able to receive the result and there is still a known access point that provides statistical data.
- Maintain a map of the number of partitions for each worker. In this case, each worker is assigned an id. If it times out, it may attempt to rejoin with the same id. The server can check if it knows about this particular worker, and make sure to assign it fewer vertices next time, even if it means splitting a partition.

The superstep is composed of the following steps:

1. The server begins the superstep, telling workers to initiate the compute step on each partition.
2. Receive a return value from each worker, consisting of:
    a. a list of messages, each of which consists of a destination vertex id, and a value (in our PageRank example, that will be a number)
    b. a *halt* flag, which is true only when all vertices on that worker have voted to halt
3. As the server receives a message, it determines which worker to send it to, then concurrently sends it to the destination worker for processing in the next superstep.
4. Once all workers have returned, if there are no pending messages and all workers returned a true *halt* flag, then the server returns the result to the client. Otherwise the server initializes a new superstep and go back to step 1.

Worker Failures:
We periodically maintain checkpoints to handle Worker Failure. Every *c* supersteps, each worker stores its vertex state and pending messages in the global DB, while the server stores current superstep metadata. In our PageRank example, *vertex state* consists of the current vertex score and its *halt* flag. We will use instrumentation to determine how often to set a checkpoint since we want to balance checkpoint cost, expected recovery cost, and mean time to failure[1].

If one or more worker dies during a superstep, the server reverts the algorithm to the last checkpoint instead of moving to the next superstep. It redistributes the partitions and messages among the remaining workers, and initializes a superstep.

Partition Distribution:
Each worker will be assigned some number of partitions to work on. To implement load-balancing, we will keep track of the performance of each machine at each superstep. We can measure the number of partitions each worker is handling, as well as the time it takes for them to finish. Using this, the server periodically modifies the partition assignment to give faster machines more work.

Issues:

- We will need to be careful with the initial distribution of partitions to make sure that we do not over or under assign resources. We have not yet decided how we will address this issue, but we have explored ideas such as round robin scheduling.

Dynamic Workers:
We use checkpoints to handle joining and departing workers. When a new worker registers, the server keeps it in pending state until the next checkpoint. Following the checkpoint's completion, the server activates the new worker, and redistributes the partitions with the new workers. By waiting for the next checkpoint instead of activating the worker immediately, we reduce the overhead of initializing workers. We treat departing workers as failed workers.

Extensions:

- Enable a departing worker to exit "gracefully". A worker-side user can signify that it will disconnect following the next superstep, in which case the server initiates a checkpoint.

Workers and TCP
The master server and workers communicate over TCP, passing json messages between them. Messages will be sent during partition assignment, superstep initiation, message delivery between both server and workers, and verification of existence of workers to detect failures.

Failures:

- If the server fails, the worker detects this through the TCP connection and terminates.
- If the worker crashes, the server handles worker failure through checkpoints, as described above.

- If a worker hangs -- for example, it is extremely slow -- the server will timeout and disconnect from it, treating it as a worker failure. At this point, the worker is free to automatically try to reconnect.

Extension:

- Use *combiners*[1] to save space in message buffers. These reduce multiple messages destined for one vertex into one message. For example, in PageRank, the following three messages for one vertex {destination: "a", value: 2}, {destination: "a", value: 3}, {destination: "a", value: 4}, would be reduced by summing the values to this one message: {destination: "a", value: 6}. Combiners are applicable when you are performing operations that are associative and commutative.

Testing

In order to easily test this system, we will need to create several automated testing scripts. Basic scripts will include one that initializes a server and workers, one that initializes clients and connects them to the system, one that sends invalid requests to the server to check basic error handling, and one that emulates a variety of failure cases: where the server dies or fails to listen for connections; a worker dies or cannot connect to the server; and where clients cannot connect to the server.

More extensive testing will include a script that emulates stress tests on the server by giving the system very large problems to handle, or very large number of workers. We will also create a script that will modify the resources of workers to test the system's ability to deal with varying processing power, connectivity, and communication delays.

We will use GoVector[3] and ShiViz[4] to visualize interactions between active workers, the server, and clients. It will be useful have access to statistics like the number of active workers, the number of active vertices, the distribution of vertices over workers, and the current superstep number. This can be done by adding an external IP to the server which can easily connect to a script that imitates a client and makes requests to retrieve the above info.

Extensions:

- It would be interesting to develop a sequential implementation of the system alongside BSP to compare performance.
- With the addition of various extensions to the system, we will also need to add test scripts that will cover elements of the extensions.
    - Mobile Computing: In this situation, different failure cases will have to be tested. For example, when a phone appears to not be responding but is still alive and working, and then rejoins with a completed computation.
    - Replication: Verifying that replication of computations remains consistent on workers and checking that the majority response is being used as the answer.

SWOT Analysis

Strengths: 1. Project is based on existing system which has been implemented in several languages. 2. Numerous papers exist describing techniques applicable to the project. 3. Complexity is mostly limited to to the server. 4. Some members are good at researching stuff.	Weaknesses: 1. Each one of us will independently be unavailable for several days throughout the project. 2. All group members began learning Go this school year.
Opportunities: 1. Currently no golang BSP library we could locate on GitHub. 2. ¾ of us are also taking 418 giving us general insights into parallel programming similar to (but not including) BSP	Threats: 1. We don't have a large number of heterogenous systems to test with 2. The department keeps hiring people with drills that follow us to every room we try to have meetings in

Timeline

February 29th:
- Proposal due

March 4th:
- Development Process finalized
- Initial tasks distributed and tracked
- Bug tracking software

March 11th:
- GlobalDB configured
- PageRank algorithm implemented
- Graph partitioning done
- Basic implementation of master server with heterogeneous worker complete
- Automated tests complete

March 18th:
- Email to schedule meeting
- Checkpoint-Restart implemented
- Worker Failures Handled
- Automated tests updated

March 21st:

- Dynamic Worker add/remove
- *Basic* implementation complete
- Testing scripts complete
- Automated tests updated

March 28th:
- Extensions 1 chosen

April 4th:
- Extensions 1 implementation complete
- Extensions 1 tests complete
- Extensions 2 chosen

April 7th:
- Extensions 2 implementation complete
- Extensions 2 tests complete
- Report draft complete
- Zero-Bug phase begins

April 11th:
- Zero-Bug Deadline
- Report completed
- Code and Report Submitted
- Pray to the programming gods

April 12th:
- Party with Cake or Pie!

References:

1. 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. http://www.dcs.bbk.ac.uk/~dell/teaching/cc/paper/sigmod10/p135-malewicz.pdf
2. https://en.wikipedia.org/wiki/Bulk_synchronous_parallel
3. https://godoc.org/github.com/drewlanenga/govector
4. http://bestchai.bitbucket.org/shiviz/index.html