Awesome Gamma

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

Mike Fink (o0p4), Dorothy Ordogh (c9f7), Graham St-Laurent (i5l8), Bruno Vacherot (p0j8)

Abstract

Large graph processing is a commonly needed task in today's society. Excess computing power means that volunteer computing can be a useful service for those needing large amounts of computation. While MapReduce has been implemented in a volunteer computing context, Pregel or Bulk Synchronous Parallel has not. We created a prototype volunteer Pregel system for processing the PageRank of graphs in Go. It provides for multiple heterogeneous workers and allows for checkpoint-restart functionality in the event of failures.

1. Introduction

Large graphs are common in the information based society of the 21st century. From the friends graph of Facebook users to citation graphs of journal papers to the connections of the web. MapReduce has become a common framework for processing large problems, including graph-based problems, via implementations like Hadoop, Spark and Disco. Additionally, the proliferation of personal computing power has made volunteer computing a successful tool for scientific computation of large datasets. These ideas have been combined in BOINC-MR implementing MapReduce on Berkeley's volunteer computing system [1]. However, MapReduce is not ideal for tackling graph problems because of the iterative nature of the problems and the issue that vertex data must be continually passed throughout the system.

Pregel, a Google implementation of the Bulk Synchronous Parallel (BSP) computing paradigm, provides a simple and effective way to process large graph problems. It is an iterative process that is designed with the mantra of "thinking like a vertex", meaning that a programmer just provides the computation that a vertex must perform each iteration. The graph is processed primarily in place and only messages between vertices must be passed through a master. Implementations of Pregel include Apache Giraph and GoldenOrb. However, we are unaware of any volunteer computing implementations of Pregel.

Our goal is to implement a volunteer computing implementation of Pregel in Go. This system is intended to be used as a tool for researchers to be able to harness a network of workers for processing large graph problems. In the context of this project, we created a prototype Pregel system that trusts its workers and processes a user-provided graph using the PageRank algorithm for ranking relevance of nodes in a graph.

Our system comprises four components:

- clients,
- a server,
- workers, and
- a database.

Clients provide graph data to the database and request that a job is created by the server. The server assigns portions of the graph to workers, who retrieve their assigned data from the database. The server then initiates and monitors the job. Failures are handled via checkpoint restarts, where workers are instructed to save their data periodically so that, in the event of a worker failure, the system can rollback to a pre-failure state and continue the computation. Workers are also assumed to be heterogeneous computing power and can be assigned more or less work by the server depending on their speed. The system produces a graph stored in the database that can be retrieved by the client.

In the sections that follow we will describe the Design of the system that was actually built (section 2), followed by details of the Implementation of the system in Go (section 3). This is followed by a description of the Evaluation we performed (section 4) and the Limitations associated with our prototype system (section 5). The paper concludes with a Discussion of challenges we experienced while building the system (section 6).

2. Design

We begin this section with a review of the Pregel programming paradigm. At a basic level, a Pregel server 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.

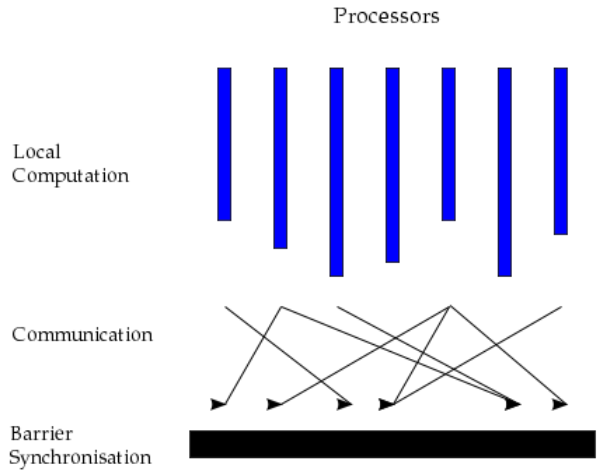
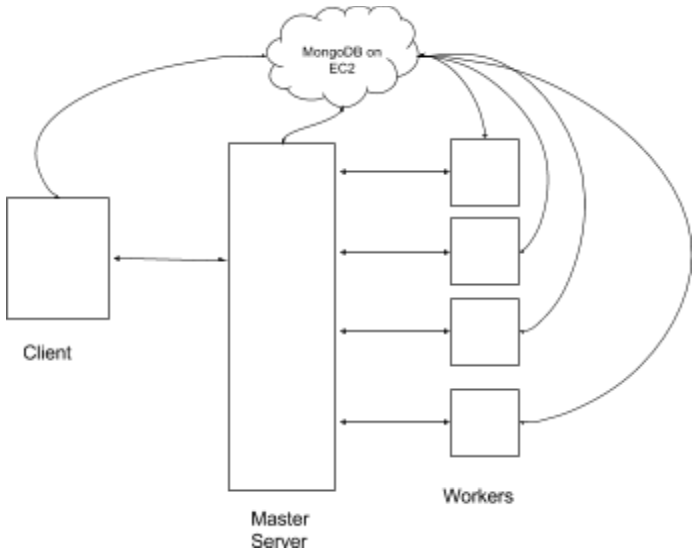


Illustration of a BSP superstep. Drawn by Discboy, specifically for the Bulk Synchronous Parallel Wikipedia page. https://en.wikipedia.org/wiki/File:Bsp.wiki.fig1.svg

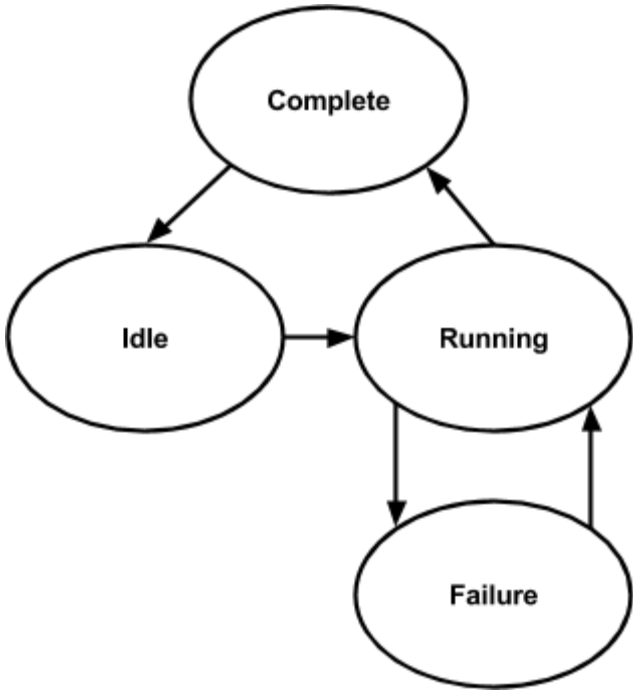
Our system is composed of three node roles: clients, the master server, and worker nodes. It also uses an external database that is currently hosted on an Amazon EC2 instance. The database provides global persistent data storage for the three components to share information. The connection to the database is built into the system such that setup is not needed in order to run the program. The connections present in the system are shown in the figure below.



System State

The state-machine below shows the various states of the system. The system begins in an idle state while waiting for a job request. It then moves to a running state while processing a job.

Optionally, if a worker fails while the job is running, the system moves to a failure state and attempts to recover back to the running state. Once a job is complete, the system moves to a complete state and informs the client before returning to an idle state to wait for additional jobs.



Client

The client is responsible for creating a new job and uploading the vertices to the database. It then sends the information to access the database in the job request to the server via an RPC request.
In our proposal the client was to send the graph to the server along with its job request. We deviated from this because it would require sending a potentially large file twice over the network rather than once directly from the client to the database. The drawback of this approach is that the client has direct access to the database, so it could interfere with the data while the job is running, intentionally or accidentally. We felt that this was appropriate in the context of the prototype.

While only one job can be run at a time, several clients can connect to the server. If more than one client connects to the server, the clients' jobs will be queued in the server. If the client fails, the server and workers will continue processing the job and store the result in the database while rejecting re-connection for clients with the same id until the job is complete. The client can retrieve the result of the job directly from the database once complete.

The external system API is limited to the client-server interaction and, in the prototype, consists only of the new job request for a client.

Master Server

The master server is responsible for accepting and managing connections from clients, receiving and managing connections from workers and running and managing jobs.

The server partitions the graph into a number of partitions, and assigns each partition to a worker. Since our model is an adaptation of Pregel, 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 vertices have voted to halt. The algorithm completes when all workers vote to halt and there are no pending messages. Because PageRank has no halting mechanism, in the prototype the algorithm is halted after 50 steps, which should be sufficient for the values to converge.

The server guarantees the correct ordering of messages while acting as the barrier. This means that each worker is sent its begin superstep command before any messages from that superstep are relayed from other workers.

If the server fails, the system is down and the job fails. The workers discover this via their TCP connection and exit. The server maintains a timeout for each superstep. If a worker's response time exceeds this timeout, then the server disconnects the worker and handles the failure as described in the Worker section below. A worker failure and disconnection is handled in the same way.

Workers

The workers are responsible for processing the value of each vertex in its assigned partition for each superstep and receiving and sending relevant messages to the server. A worker sends and receives messages for the next superstep while processing the current one. Once the superstep is complete it sends a step completed message to the server.

For our prototype, each vertex is a PageRank vertex and so the PageRank of each is calculated every superstep. The capability to extend this to other algorithms via other types of vertices is built into the system, but no other types were created for the prototype.

We use a checkpoint-restart procedure for worker failure recovery. Every *c* supersteps, the server requests that each worker stores its vertex state and pending messages in the global DB.

If one or more worker dies during a superstep, the server reverts the algorithm to the most recent checkpoint instead of moving to the next superstep. It redistributes the partitions and messages among the remaining workers, involving a request that the workers load from the database and then initializes a superstep.

3. Implementation

Our system comprises three programs and a cloud-based database. The programs are for a client, the server and a worker. Each is command-line based and written entirely in Go.

The database consists of an Amazon EC2 instance running a community version of MongoDB. Jobs are stored as tables consisting of all the vertices and associated information for the graph.

Communication between the client and the server is over RPC for job requests. Communication between the server and the workers comprises a custom protocol built on TCP. This was selected because we required both two-way communication and out-of-band communication between the workers and the server. A worker must be able to provide a stream of messages back to the server during a superstep, and this is difficult to achieve over an RPC connection. TCP was selected in order to provide easy failure detection and to guarantee message ordering over a single connection. The server and worker communicate with 8 types of messages, 3 from the server to the worker, 4 from the worker to the server and 1 used in both directions. The server may assign vertices to a worker (which doubles as request to load from a checkpoint), request that a worker save a checkpoint, and request that a worker perform a superstep. The worker's messages respond to each of these commands with success or failure. The shared messages comprise vertex to vertex messages passed from worker to worker through the server.

Our code contains one unit tests file, dedicated to the server's management of the distribution of vertices across workers. We also created a Python script to sequentially determine the PageRank using the traditional matrix calculation; we compared our program's output to the sequential script to determine if it was successful. We also have a Python script that automatically generates graph files, so we can easily run tests on unlimited arbitrary graphs.

GoVector on the client for its communications with the server; it is used by the server for its communications with the client and between the server and workers; and it is used on the worker for communications with the server. It is not used for communications with the database, this is because the database is assumed to be always available would not provide interesting information.

4. Evaluation

There are two main ways that we evaluated our system: looking at logs, and comparing our results to the sequential result from a matrix calculation. Looking at logs, we could check if the appropriate workers were performing the appropriate operations in the correct sequence. Since Pregel is Barrier Synchronized, the sequence of events on each worker is predictable and easy

to follow. We constructed our client program so that when the server informs it of completion, it creates an output file with the PageRanks for each vertex of the input graph. By comparing these results to the file output by the sequential program on small graphs using a tool like diff, we could verify the correct behaviour.

That being said, we actually ran into many integration issues involving the database, and discovered unresolvable bugs when downloading the resulting graph, so we were unable to actually do the comparing testing method.

5. Limitations

The system is limited in various ways as mentioned in places above. One primary limitation is that the server trusts the workers to provide accurate data. This is an issue for a volunteer computing system because you do not have control over the workers that attempt to join. Two ways to mitigate this issue are to replicate work from workers and compare the messages being sent on the server, or to periodically allocate test jobs in order to spot check results from workers. Both of these introduce additional overhead to the system, slowing it down (as well as adding additional complexity).

Another limitation of the project is that we expect its usefulness to be limited to extremely large files. This is because the time taken for a worker to process all of the vertices in its partitions in a single superstep needs to be proportional to the time it takes to send all the vertex to vertex messages through the server. We expect that it would require significant optimization to determine useful system parameters for graph distribution and communication.

Related to this limitation is the decision to have a central server pass messages between all nodes in the system. The master server will act as a bottleneck for the system limited by its bandwidth and processing power. The tradeoff for this is a simpler prototype that could actually be completed. The alternative would be to allow workers to have a list of other workers processing the job and allow them to send messages directly between one another, only having the step completion messages pass through the master server as a barrier. This introduces additional problems of straggling messages as well as great complexity.

6. Discussion

While trying to implement global persistent data storage, we began using the free tier of Amazon’s DynamoDB. After using the Go AWS SDK and running a number of tests on small and large datasets we found that uploading all the vertices to start the job took more than two minutes because DynamoDB was throttling requests made to it. Maximizing the read and write capacity of the table ended up exceeding the free tier and costing money, so we decided to try installing a local version of DynamoDB on an EC2 instance. EC2 allows far more requests than is required for our system to run, so we thought this would be the better approach.

Unfortunately, the local database still throttled the requests and so was excessively slow. This meant we had to fall back to what one of members knew, using MongoDB on the EC2 instance.

Our team ran into issues with the integration of our system. We underestimated the workload required to design and implement a modular system of this size, which ultimately limited the success of our project. We feel however, that we had an ambitious and tried to tackle the problem in a good way.

Allocation of Work

Mike Fink worked on the worker implementation and report; Bruno Vacherot worked on the communication between client, server, and worker; Graham St-Laurent worked on testing, debugging, and server implementation; and Dorothy Ordogh worked on database schema setup and communication.

References

1. Fernando Costa, Luis Veiga, and Paulo Ferreira. BOINC-MR: MapReduce in a Volunteer Environment. http://www.gsd.inesc-id.pt/~pjpf/coopis-BOINC-MR-2012.pdf