Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert,
Ilan Horn, Naty Leiser, Grzegorz Czajkowski
Google, Inc.
{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming—*Distributed programming*; D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.13 [Software Engineering]: Reusable Software—*Reusable libraries*

General Terms: Systems, Design, Algorithms

Keywords: Distributed computing, graph algorithms

1. INTRODUCTION

Many practical computing problems concern large graphs. Standard examples include the Web graph and various social networks. The scale of these graphs—in some cases billions of vertices, trillions of edges—poses challenges to their efficient processing. Despite the ubiquity of large graphs and their commercial importance, we know of no scalable general-purpose system for implementing graph algorithms in a distributed environment. To address distributed processing of real-life graphs, we defined a model of computation and realized it through a scalable and fault-tolerant system called Pregel, with an expressive and flexible API.

The high-level organization of Pregel programs is inspired by Valiant's Bulk Synchronous Parallel model. Pregel computations consist of a sequence of iterations, called *supersteps*. During a superstep the framework invokes a user-defined `Compute()` function for each vertex, conceptually in parallel. The function specifies behavior at a single vertex v and a single superstep S. It can read messages sent to v in superstep $S-1$, send messages to other vertices that will be received at superstep $S+1$, and modify the state of v and its outgoing edges. Messages are typically sent along outgoing edges, but a message may be sent to any vertex whose identifier is known. A program terminates when all vertices declare that they are done.

The input and output are both directed graphs. They are often but not always isomorphic, because vertices and edges can be added and removed during computation. User-defined *handlers* are applied to resolve conflicts for concurrent mutations.

The vertex-centric flavor of programming in Pregel is similar to the MapReduce model in that programmers focus on a local action, processing a single item at a time, which the system then lifts to computation on a large dataset.

The synchronicity of the Pregel model simplifies writing correct programs and simplifies reasoning about the interactions of computation, message passing and mutation of topology. Pregel programs are inherently free of deadlocks and data races common in asynchronous systems. At the same time the performance should be competitive with an asynchronous system, because typical graph computations have high parallel slack (many more vertices than machines).

The basic computing model of Pregel is enhanced by several mechanisms to improve performance and usability. They include combiners, an optimization to reduce network traffic, and aggregators, a simple mechanism for monitoring and global communication.

By design the model is well suited for distributed implementations: it doesn't expose any mechanism for detecting order of execution within a superstep, or any communication mechanism other than communication from superstep S to superstep $S+1$. Vertices are grouped into *partitions*; programmers can optionally define the mapping to minimize network traffic and the impact of the associated latency. For example, a typical heuristic employed for the Web graph is to collocate vertices representing pages of the same site. A master assigns partitions to workers, directs them to perform `Compute()` functions, and then synchronizes workers at a barrier. Fault tolerance is achieved through checkpointing.

2. EXPERIMENTAL RESULTS

We conducted an experiment solving single-source shortest paths using a simple Belman-Ford algorithm expressed in Pregel. As input data, we chose a randomly generated graph with a log-normal distribution of outdegrees; the graph had 1B vertices and approximately 80B edges. Weights of all edges were set to 1 for simplicity. We used a cluster of 480 commodity multi-core PCs and 2000 workers. The graph was partitioned 8000 ways using the default partitioning function based on a random hash, to get a sense of the default performance of the Pregel system. Each worker was assigned 4 graph partitions and allowed up to 2 computational threads to execute the `Compute()` function over the partitions (not counting system threads that handle messaging, etc.). Running shortest paths took under 200 seconds, using 8 supersteps.

This is not the best shortest-path runtime achievable with the Pregel framework. Rather, it shows that a non-expert in distributed computing can easily use Pregel to implement an algorithm and execute it on a cluster, for a large-scale graph beyond the reach of single-machine solutions, and obtain runtimes comparable to those from a custom distributed computing solution.

PODC'09 & SPAA'09, August 10–13, 2009, Calgary, Alberta, Canada.
ACM 978-1-60558-396-9/09/08 & ACM 978-1-60558-606-9/09/08.