Large Graph Processing in the Cloud

Rishan Chen[†‡] Xuetian Weng[†‡]
[†]Peking University
{crs,wengxt}@pku.edu.cn

Bingsheng He[‡] Mao Yang[‡]
[‡]Microsoft Research Asia
{savenhe,maoyang}@microsoft.com

ABSTRACT

As the study of graphs, such as web and social graphs, becomes increasingly popular, the requirements of efficiency and programming flexibility of large graph processing tasks challenge existing tools. We propose to demonstrate *Surfer*, a large graph processing engine designed to execute in the cloud. Surfer provides two basic primitives for programmers – *MapReduce* and *propagation*. MapReduce, originally developed by Google, processes different key-value pairs in parallel, and propagation is an iterative computational pattern that transfers information along the edges from a vertex to its neighbors in the graph. These two primitives are complementary in graph processing. MapReduce is suitable for processing flat data structures, such as vertex-oriented tasks, and propagation is optimized for edge-oriented tasks on partitioned graphs.

To further improve the programmability of large graph processing, Surfer consists of a small set of high level building blocks that use these two primitives. Developers may also construct custom building blocks. Surfer further provides a GUI (Graphical User Interface) using which developers can visually create large graph processing tasks. Surfer transforms a task into an execution plan composed of MapReduce and propagation operations. It then automatically applies various optimizations to improve the efficiency of distributed execution. Surfer also provides a visualization tool to monitor the detailed execution dynamics of the execution plan to show the interesting tradeoffs between MapReduce and propagation. We demonstrate our system in two ways: first, we demo the ease-of-programming features of the system; second, we show the efficiency of the system with a series of applications on a social network. We find that Surfer is simple to use and is highly efficient for large graph-based tasks.

Categories and Subject Descriptors

C.2.4 [**Computer-communication networks**]: Distributed systems—*Distributed databases*; H.2.4 [**Database Management**]: Systems—*Graph Processing*

SIDMOD'10 June 6–11, 2010, Indianapolis, Indiana, USA

General Terms

Algorithms, Measurement, Performance

Keywords

Graph processing, distributed systems, MapReduce, propagation

1. INTRODUCTION

Large graph processing has become popular for various applications such as for studying web and social networks [6, 7, 11]. Most processing tasks are batch tasks in which vertices and edges of the entire graph are accessed. Examples include PageRank [10], reverse link web graph, two-hop friend list, social network influence analysis [12] and recommender systems [1]. Due to diversity of user requirements, these tasks tend to be highly customized with various user-defined code. We have developed Surfer, a large graph engine that automatically runs on a large number of machines in the cloud, and enables users to efficiently and conveniently develop such large graph processing tasks.

We began by investigating whether the popular MapReduce [3] algorithm is sufficient for supporting large graph processing tasks. The data model in MapReduce is flat, making it ideal for handling the vertex oriented tasks on a large graph e.g. filtering the vertices with a certain degree. However, we found that the obliviousness of MapReduce to the graph structure leads to huge network traffic in edge-oriented tasks, creating a bottleneck in the system. For example, if we want to compute the two-hop friend list for each account in the MSN social networks, every vertex must first send its friends to each of its neighbors, then each vertex combines the friends lists of its neighbors. Implemented with MapReduce, this operation results in huge network traffic due to the flat key-value data model which does not reflect the intrinsic structure of the graph.

In analyzing the edge-oriented tasks we observed access patterns in which information is transferred along each edge from a vertex to its neighbors in the graph. The two-hop friend list computation is an example of this pattern. Another example is PageRank – in each iteration, the rank of a page is partially given to its outgoing pages along the links, and then the target page combines all the awarded partial ranks.

We introduce the *graph propagation* primitive to make these patterns more intuitive to specify and easier to support by the task engine. To use graph propagation, the user defines two functions – $transfer$ and $combine$. $Transfer$ is used to export information from a vertex to its neighbors, while $combine$ is used to aggregate the received information at a vertex. This primitive is easily parallelized across multiple machines. We adopt the graph partitioning algorithm [8, 9] to divide the large graph into many partitions of similar sizes. Each machine may hold a number of graph par-

titions, and perform propagation locally before coordinating with other machines. Since the number of edges that span across machines is a good indicator on the amount of network traffic during the transfer stage, it should be minimized through the graph partitioning. Thus, propagation performs computation on the partitioned graph, and exploits the locality of graph partitions.

The graph propagation primitive is insufficient on its own. For example, it is non-trivial to express a vertex-oriented operation, such as filtering of vertices, using propagation alone. We found that a combination of MapReduce and propagation is necessary to effectively express and to efficiently compute most graph based tasks.

Supporting both MapReduce and propagation, the Surfer system consists of a job scheduler, a job manager, and many slave nodes. The job scheduler maintains the cluster membership and coordinates resource scheduling. The job manager takes a user's job as input, performs transformation and optimization on the job, and executes the job by dispatching the corresponding tasks to slave nodes. The slave nodes store graph partitions and execute the tasks assigned to them by the job manager.

Our system demonstration will show how users develop large graph processing programs using Surfer. It will also highlight intrinsic optimizations and execution dynamics within the system through visualization. Our demo makes the case that Surfer provides users with the necessary intuition to understand complex computations on large graphs. Our demo is organized into three parts:

- Demonstrating a drag-and-drop GUI for developing the logical execution plan for large graph processing. The GUI allows users to develop their own customized high-level logical operations, and to use these operations to construct an execution plan.
- Visualizing the transformation and optimization processes that convert a logical execution into a physical plan. This reveals the underlying mechanism of Surfer on the job manager node.
- Visualizing the resource utilization and execution progress. The visualization helps users to monitor the execution dynamics, such as whether the resources are under-utilized, how the plan is executed on multiple machines, and to identify potential performance bottlenecks.

Throughout the demonstration, we show that (1) the drag-and-drop GUI improves the programmability of large graph processing, and (2) query optimizations are effective in reducing redundancy, improving computation efficiency and storage locality.

2. SYSTEM OVERVIEW

We first briefly introduce the propagation primitive and the Surfer architecture. Next, we describe our visualization tool for job construction, optimization and execution.

2.1 Propagation

Surfer provides the *propagation* primitive to facilitate developers to implement their custom logics. To use graph propagation, the developer defines two functions – *transfer* and *combine*. *Transfer* is used to export data from a vertex to its neighbors, while *combine* is for processing the received data at a vertex.

Iterative propagation is to transfer the information of each vertex to its neighbor vertices iteratively. At each iteration, the information transfer is occurred along all the edges. This information flow consists of the basic pattern on traversing the graph in parallel. A lot of common graph applications and algorithms, such as PageRank and two-hop friend list, are expressible with this primitive.

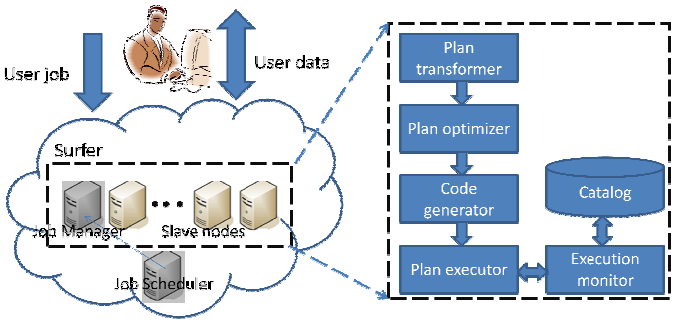


Figure 1: The system architecture of Surfer

Function *transfer* defines how the information is transferred along an edge, and *combine* defines how the information from its neighbors is combined at each vertex. In particular, *transfer* takes a pair of a vertex and a value as input, and outputs pairs of a (neighbor) vertex and a value each. *combine* takes the pair of a vertex and all the values associated with the vertex generated in *transfer*, and outputs a pair of a node and a value. The signatures of these two functions are as follows.

$$transfer\text{: } (v, v') \rightarrow (v', value), \text{ where } v' \text{ is } v\text{'s neighbor.}$$
$$combine\text{: } (v, \text{bag of } value) \rightarrow (v, value').$$

Both user-defined functions are operations on vertices and edges. Surfer executes an iteration of propagation in two steps: 1) the Transfer stage: Surfer calls *transfer* on each vertex and its neighbor vertices, and generates the intermediate results; 2) the Combine stage: Surfer calls *combine* on the intermediate results generated in the Transfer stage.

2.2 Surfer

Figure 1 shows the system architecture of Surfer. Surfer takes jobs from the user as input, and automatically executes each job across multiple machines.

Surfer has five major components, namely a plan transformer, a plan optimizer, a code generator, a plan executor and an execution monitor. These components are responsible for the five steps in a task execution.

Step 1. The plan transformer transforms a task into a DAG (Directed Acyclic Graph). Each DAG node represents an occurrence of MapReduce or graph propagation.

Step 2. The plan optimizer applies various optimization rules to the logical plan. We adopt optimization rules from databases to reduce the intermediate data size. The optimization rules include 1) removing unnecessary nodes, 2) identifying common computation nodes, and 3) reordering nodes.

Step 3. The optimized logical plan is transformed into a physical plan. Each node in the physical plan can either be a MapReduce or a propagation operation. The optimizer chooses to use MapReduce for vertex-oriented tasks, and propagation otherwise. The optimizer further adopts physical optimization techniques [5], such as pipelined execution for multiple DAG nodes.

Step 4. After the plan is optimized, the code generator generates the executable for a distributed execution.

Step 5. The plan executor performs the execution on the slave nodes. During the execution, the execution monitor records the resource utilization and estimates the execution progress of the job. The optimizations in Steps 2 and 3 are rooted in traditional database query optimizations. While database optimization techniques are

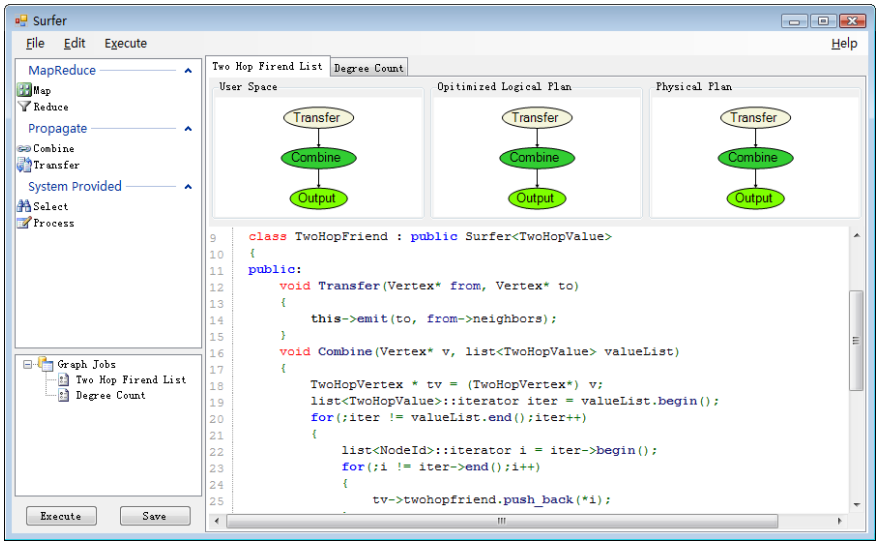


Figure 2: The main execution flow window

effective on SQL-like systems [2, 4], they need to be revisited in the context of graph processing.

A catalog is used to store the execution dynamics produced by the execution monitor. For example, the selectivities of filtering conditions are stored to help estimate the cost when the same filtering conditions are used.

The large graph are partitioned, and are evenly stored in the slave nodes. Since graphs in the real world are usually sparse, we choose adjacency lists as our storage model for efficiency, in contrast with the matrix format in PEGASUS [7].

Additionally, Surfer provides jobs with fault tolerance to machine failure. Whenever a machine crashes, the task is restarted on another machine. For large graph processing jobs, this is an important feature as many weeks worth of computation may be lost due to a job failure.

2.3 Visualization tools

The visualization tool is a graphical user interface (GUI) that facilitates users to develop their tasks, and dynamically displays the runtime performance and the state of task execution in the system. The GUI is implemented in C#.

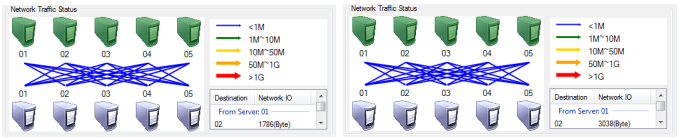
Figure 2 illustrates how a graph processing task is constructed with the GUI.

Job construction. To further improve the programmability of graph processing, Surfer integrates high-level building blocks such as `Select` and `Process`. These building blocks can be applied to vertices, edges, and custom constructs specified by the developer.

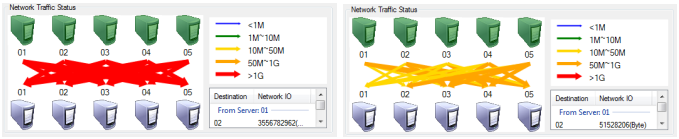
In the GUI, to add a building block to the current logical query plan, developers need to simply 1) select the input type for the building block, and 2) drag the corresponding building block into the correct position as a *logical node*, and then input the necessary information such as user-defined functions. The user can further link two logical nodes according to their data dependency. Users may also input their custom code to execute on the logical node in the editor below.

Optimization. After users finish setting up the logical nodes, the logical plan is generated, and the optimized logical plan is built as well.

Execution. The visualization tool in Surfer shows the execution dynamics per machine or per task. Figure 3 shows a screen shot of



(a) MapReduce-Degree Count (b) Propagation-Degree Count



(c) MapReduce-Two Hop (d) Propagation-Two Hop

Figure 4: Screen shots: Network traffic of degree count and two-hop friend list computation

visualizing execution dynamics of a Transfer stage in a propagation operation.

The visualization tool allows users to select the stage they want to investigate in three panels. Panel 1 shows task execution on each machine. From the task execution dynamics, users can determine whether the system works well, for example, whether a certain task is stuck on a machine. Panel 2 at the bottom displays the network traffic between machines. Panel 3 on the right monitors resource of a specific machine, such as CPU, memory, disk IO and network IO.

3. DEMONSTRATION

We use the visual tool to perform some case studies in large-graph processing.

3.1 Demo setup

We have two approaches for showing our demonstration. One is to run Surfer on a real cluster of 32 nodes, and show our demo via remote desktop in Windows; the other is that we will set up a virtual cluster on a PC consisting of five virtual machines. The former approach is our preferred approach, and the latter is for backup in

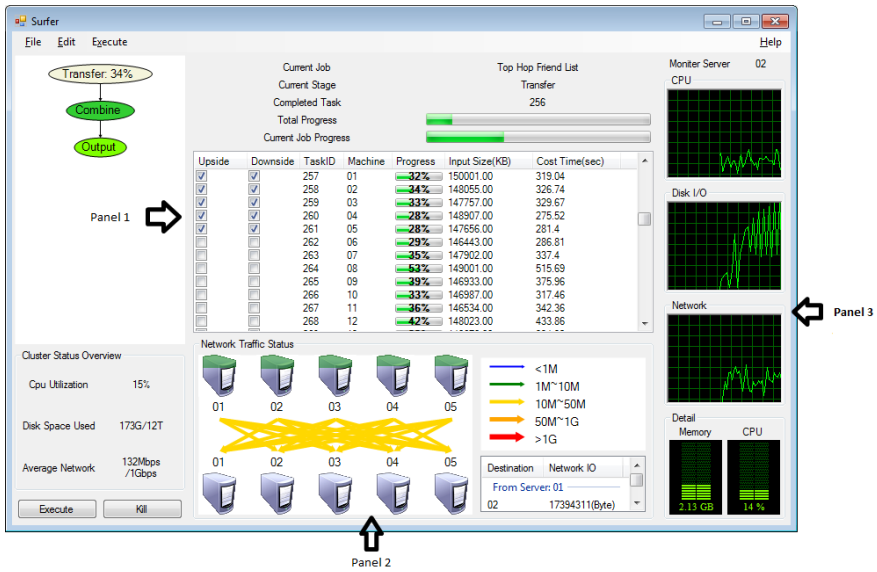


Figure 3: Visualizing the execution dynamics

case where the network connection from the demo place to our data center does not work during the demonstration.

Each node in our cluster consists of one Intel Quad CPU, with 8 GB RAM and two SATA disks. The data set is a portion of the anonymized MSN social graph. The total data set size is 120GB with around 500 million vertices and 30 billion edges.

In our demonstration, we will also demonstrate a number of common social network applications. Two examples are “degree count" that is a vertex-oriented task for calculating the number of vertex neighbors in the network, and “two-hop friend list" that is an edge-oriented task for collecting the neighbors’ neighbors of each person.

3.2 Runtime performance results

In our demonstration, we will show the runtime performance results of Surfer. An example snapshot is shown in Figure 3, where the execution dynamics of two-hop friends are visualized.

Additionally, our demonstration will compare the performance of constructing the same applications using our two different primitives (MapReduce or Propagation). As an example, Figure 4 shows some screen shots of the two example applications using MapReduce and propagation. Since propagation takes advantage of partitioned graph structure and avoids unnecessarily data partitioning, its network traffic is much smaller than MapReduce in the two-hop friend list computation. We also find that the network traffic is almost the same for the two primitives in the degree count calculation. MapReduce has a lower loading time (the result is not shown). The visualization tool clearly demonstrates the interesting tradeoffs in the all-to-all network traffic between the two types of tasks with MapReduce and propagation.

Acknowledgement

We would like to thank Bo Peng, as well as the anonymous reviewers, for their insightful comments.

4. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowl. and Data Eng.*, 17(6):734–749, 2005.

[2] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2), 2008.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[4] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou. Wave computing in the cloud. In *HotOS*, 2009.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[6] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. HADI: Fast diameter estimation and mining in massive graphs with hadoop. Technical Report CMU-ML-08-117, Carnegie Mellon University, 2008.

[7] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system - implementation and observations. In *ICDM*, 2009.

[8] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing ’96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 1996. IEEE Computer Society.

[9] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, 1998.

[10] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[11] Pregel in Google. *http://googleresearch.blogspot.com/2009/06/large-scale-graph-computing-at-google.html*.

[12] H. T. Welser, E. Gleave, D. Fisher, and M. Smith. Visualizing the signatures of social roles in online discussion groups. *The Journal of Social Structure*, 2(8), 2007.