On the Efficiency and Programmability of Large Graph Processing in the Cloud

Rishan Chen[1,4] Xuetian Weng [1,4] Bingsheng He [1,2]
Mao Yang[1] Byron Choi[3] Xiaoming Li [4]
[1]*Microsoft Research Asia* [2]*Nanyang Technological University*
[3]*Hong Kong Baptist University* [4]*Beijing University*

ABSTRACT

As the study of large graphs over hundreds of gigabytes becomes increasingly popular in cloud computing, efficiency and programmability of large graph processing tasks challenge existing tools. The inherent random access pattern on the graph generates significant amount of network traffic. Moreover, implementing custom logics on the unstructured data in a distributed manner is often a pain for graph analysts. To address these challenges, we develop *Surfer*, a large graph processing engine in the cloud. Surfer resolves the bottleneck of network traffic with graph partitioning, which is specifically adapted to the network environment of the cloud. To improve the programmability, Surfer provides two basic primitives as building blocks for high-level applications – *MapReduce* and *propagation*. Surfer implements both primitives with automatic optimizations on the partitioned graph. We implement and evaluate Surfer with common graph applications on the MSN social network and the synthetic graphs with over 100GB each. Our experimental results demonstrate the efficiency and programmability of Surfer.

1 INTRODUCTION

Large graph processing has become popular for various applications on the increasingly large web and social networks [12, 13]. Due to the ever increasing size of graphs, application deployments are moving from small-scale HPC servers [16] towards the cloud with massive storage and parallelism [20, 13]. Most processing tasks in these graph applications are batch operations in which vertices and/or edges of the entire graph are accessed. Examples of these tasks include PageRank [19], reverse link graphs, two-hop friend lists, social network influence analysis [21], and recommender systems [1]. The inherent random access pattern on the graph generates significant amount of network traffic. Moreover, graph applications tend to be highly customized according to user requirements. Therefore, a large graph processing engine with high efficiency and programmability is a must. This motivates us to develop *Surfer* to enable developers to easily implement their custom logics, and to adapt to the cloud environment for efficiency.

The state-of-the-art studies on building such an engine is to investigate whether current distributed data-intensive computing techniques in the cloud [5, 10] are sufficient for the programmability and efficiency. Most of these studies are built on top of MapReduce, originally proposed by Google, which has become a popular computing paradigm for processing huge amounts of data in a massively parallel way. The representative systems include DISG [25] and PEGASUS [12, 13]. With the mature software stacks, they can handle very large graphs. For example, the experiments of PEGASUS handle a web graph with up to 7 billion edges.

While leveraging existing techniques for graph processing is clearly a step in the right direction, the efficiency can be hindered by their obliviousness to the graph structure. The underlying data model in MapReduce, GFS [6], 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 other tasks. 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 friend lists of its neighbors. Implemented with MapReduce, this operation results in huge network traffic by shuffling the vertices without the knowledge of the graph structure.

A traditional way of reducing data shuffling in distributed graph processing is graph partitioning [18]. Graph partitioning minimizes the total number of cross-partition edges among partitions in order to minimize the data transfer along the edges. Thus, Surfer stores the graph into partitions, as opposed to a flat storage in MapReduce. However, adopting MapReduce on the partitioned graph does not solve all the problems in the efficiency and programmability.

First, storing and processing the partitioned graph needs to be revisited in the cloud network environment. The cloud network environment is different from those in previous studies [16, 14], e.g., Cray supercomputers or a small-scale cluster. The network bandwidth is often the same for every machine pair in a small-scale cluster. However, the network bandwidth of the cloud environment is uneven among different machine pairs. One example of the bandwidth unevenness is caused by the network topology in the cloud. Current cloud infrastructures are often based on tree topology [7]. Machines are first grouped into *pods*, and then pods are connected higher-level switches. The intra-pod bandwidth is much higher than the cross-pod bandwidth.

Second, MapReduce can not fully exploit the data locality on the partitioned graph, for example, data shuffling after the Map stage is usually hash partitioning, which is oblivious to the graph partitions. Even worse, handling on the partitioned graph increases the programming complexity. Taking a graph partition as input to MapReduce, developers have to handle tedious implementation details such as graph partitions and boundaries in their custom code.

These problems motivate us to look for the right scheme for partitioning and storing the graph, and the right abstraction on graph processing with partitioned graph.

As for graph partitioning, we develop a bandwidth aware graph partitioning algorithm to adapt different bandwidth requirements in the process of graph partitioning to the network bandwidth unevenness. The algorithm models the machines used for graph processing as a complete undirected graph (namely *machine graph*): the machine as vertex, and the bandwidth between any two machines as the weight. The bandwidth aware algorithm recursively partitions the data graph with bisection, and partitions the machine graph with bisection correspondingly. The bisection on the data graph is performed with the corresponding set of machines selected from the bisection on the machine graph. The recursion continues until the data graph partition can fit into the main memory. This method adapts the number of cross-partition edges in the recursion of partitioning the data graph to the aggregated amount of bandwidth among machines in the recursion of partitioning the machine graph.

To exploit the locality on graph partitions, we introduce iterative *propagation* to abstract the access pattern of the batch processing on the graph. This idea is inspired by brief propagation [22], a popular iterative graph access pattern on the graph. In an iteration, information is transferred along each edge from a vertex to its neighbors in the graph. Propagation represents the common access pattern among many graph applications, such as the two-hop friend computation and PageRank.

Surfer provides the *propagation* primitive to facilitate developers to implement their custom logics. To use the primitive, developers define 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. We further develop automatic optimizations to exploit the data locality of graph partitions.

Overall, Surfer supports both MapReduce and propagation on partitioned graph. Developers can choose either primitive to implement their applications. We have evaluated the efficiency of the two implementations on a MSN social network and synthetic graphs of over 100GB each. These graphs are over five times much larger than those in the previous studies [13]. The experimental results demonstrate that 1) our bandwidth aware graph partitioning scheme improves the partitioning performance by 39–55% under different simulated network topologies, and improves the graph processing by 6–29%; 2) our optimizations in propagation reduce the network traffic by 30–95%, and the total execution time by 30–85%; 3) propagation outperforms MapReduce with a performance speedup of 1.7–5.8 times, and with much fewer code lines by developers.

The rest of the paper is organized as follows. We review the related work on large-scale data processing in the cloud, graph processing and graph partitioning in Section 2. Section 3 describes our architectural design of Surfer. We present our distributed graph partitioning algorithm in Section 4, followed by graph propagation in Section 5. We present the experimental results in Section 6, and conclude this paper in Section 7.

2 Preliminary and Related Work

We review the preliminary and the related work that is closely related to this study.

Cloud network. A cloud consists of tens of thousands of connected commodity computers. Due to the significant scale, the network environment in the cloud differs to the small-scale cluster. The current cloud practice is to use the switch-based tree structure to interconnect the servers [7]. At the lowest level of the tree, servers are placed in a pod (typically 20-80 servers) and are connected to a pod switch. At the next higher level, server pods are connected using core switches, each of which connects up to a few hundred server pods. With the tree topology, a two-level tree can support thousands of servers. The key problem of the tree topology is the network bandwidth of any machine pair is not uniform, depending on the switch connecting the two machines. The average inter-pod bandwidth is much lower than the average intra-pod bandwidth [10, 23]. Moreover, as commodity computers evolve, the cloud evolves and becomes heterogenous among generations [24]. For example, current main-stream network adaptors provide 1Gb/sec, and the future ones with 10Gb/sec. These hardware factors induce the unevenness in the network bandwidth between any two machines in the cloud.

The unique network environment in the cloud motivates advanced network-centric optimizations in cloud systems (such as multi-level data reduction along the tree topology [5, 23]) and scheduling techniques [11]. More details on these cloud systems can be found in Appendix A.1. This paper develops network-centric optimizations for partitioning, storing and processing a large graph.

Large graph processing. Batched processing on large graphs have become hot recently, due to the requirement on mining and processing those large graphs. Examples include PageRank [19] and triangle counting [21]. Surfer is designed to handle the batched graph processing applications.

There is some related work on specific tasks on large graph processing in the data center [12, 25]. MapReduce [5] and other systems such as DryadLINQ [23] were applied to evaluate the PageRank for ranking the web graph. HADI [12] and PEGASUS [13] are two recent graph processing implementations based on Hadoop. HADI [12] estimates the diameter of the large graph. PEGASUS [13] supports graph mining operations with a generalization of matrix-vector multiplication. DisG [25] is an ongoing project for the web graph reconstruction using Hadoop. Pregel [20] is a project within Google for large-scale graph processing, inspired by the bulk synchronous parallel model. However, there is no sufficient public information about Pregel for comparison. None of these engines utilizes graph partitions or extracts the access pattern specially for the batched processing on the graph, as Surfer does.

Graph partitioning. We denote a graph to be $G = (V, E)$, where V is a (finite) set of vertices, and E is a (finite) set of edges representing the connection between two vertices. The graph can be undirected or directed. This study focuses on directed graphs.

We use G_i to denote a subgraph of G and V_i to denote the set of vertices in G_i. We define a (non-overlapping) partitioning of graph G, $P(G)$, to be $\{G_1, G_2, ..., G_k\}$, where $\forall\ i \in [1...k]$, $k \leq |V|$, $\cup_{i=1}^{k} V_i = V$, $V_i \cap V_j = \emptyset$, where $i \neq j$. We define an edge to be an *inner-partition edge* if both its source and destination vertices belong to the same partition, and a *cross-partition edge* otherwise. A vertex is an *inner vertex* if it is not associated with any cross-partition edge. Otherwise, the vertex is a *boundary vertex*.

Graph partitioning is a well-studied problem in combinatorial optimization. The problem optimizes an input objective function. The input objective function in this study is to minimize the number of cross-partition edges with the constraint of all partitions with similar number of edges. This is because, the total number of cross-partition edges is a good indicator for the amount of communication between partitions in distributed computation. It is an NP-complete problem [17]. Various heuristics [17, 18] have been proposed to find an optimal partitioning. Karypis et al. [16, 14] proposed a parallel algorithm for multi-level graph partitioning, with a bisection on each level. We refer readers to Appendix A.2 for more details on graph bisection. Metis and ParMetis are serial

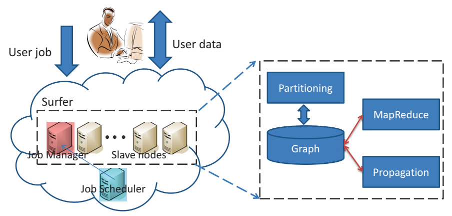


Figure 1: The system architecture of Surfer

and parallel software packages respectively for partitioning graphs [18].

Existing distributed graph partitioning algorithms, e.g., ParMetis [18], are suboptimal in the cloud. In particular, they do not consider the unevenness of the network bandwidth in the cloud. While they have demonstrated very good performance on shared-memory architectures [16], unevenness of network bandwidth results in deficiency in partitioning itself and further in storing and processing graph partitions. For example, the two partitions with a relatively large number of cross-partition edges should be co-located within a pod, instead of storing in different pods. Thus, we adapt the parallel version of graph partitioning [14] to the network environment in the cloud.

3 System Overview

Surfer allows developers to program data analysis tasks with MapReduce and propagation as building blocks. In particular, developers implement their custom logics in the user-defined functions provided in MapReduce and propagation. The distributed execution of these user-defined functions is automatically handled by Surfer. Developers do not need to worry about the underlying details on the distributed execution in the cloud.

The design of Surfer aims at scaling in a large number of machines in the cloud. Figure 1 shows the system architecture of Surfer. The system consists of a job scheduler, a job manager, and many slave nodes. A job can consist of multiple tasks implemented with MapReduce or propagation. The job scheduler maintains the cluster membership and coordinates resource scheduling. The job manager takes a user's job as input, and executes the job by dispatching the corresponding tasks to slave nodes. Each slave node stores a part of graph data and executes the tasks assigned by the job manager. We refer readers to Appendix B for more details on the Surfer architecture.

The data graph is divided into many partitions with similar sizes. We present our distributed graph partitioning algorithm in Section 4. The partitions are stored on slave machines. For reliability, each partition has three replicas on different slave machines. The replication protocol is the same as that in GFS [6].

Surfer uses the adjacency list storage as graph storage. The format is $< ID, d, neighbors >$, where ID is the ID of the vertex, d is the degree of the vertex, and $neighbors$ contains the vertex IDs n_0, ..., n_{d-1} of the neighbor vertices.

In the following, we introduce MapReduce and propagation implementations from the developer's perspective.

3.1 MapReduce

With the partitioned graph, Surfer provides the map function with a graph partition as input, in order to exploit the data locality within the graph partition. Thus, developers can exploit the locality of graph partitions in the map function so that data reduction can be performed at the granularity of graph partition. This can improve the overall performance with reduced network traffic. However, this reduces the programmability with MapReduce. Handling graph partitions such as data reductions requires tedious programming and lays the burden of implementing such optimizations on developers.

Another issue is that the Reduce stage cannot take the advantage of partitioned graphs. Since the Reduce stage requires a traditional data shuffling (usually hash partitioning) across the network, the Reduce stage is still oblivious of graph partitions, and results in a large amount of network traffic.

We use network ranking as a case study on comparing the implementation with propagation and MapReduce. Network ranking (*NR*) is to rank the vertices in the graph using PageRank [19]. Ranking the social network is useful in assigning reputation to individuals and finding the influential persons [9]. The PageRank calculation is iterative: the rank of a vertex at the current iteration is calculated based on its rank and the ranks of its neighbors in the previous iteration. Within an iteration, the access pattern of PageRank matches propagation. Each vertex distributes a part of its rank to all vertices it connects to. After the distribution, each vertex adds up its awarded partial ranks. The basic way of adding up the partial ranks for vertex v is $PR(v) = (1-d)/N + d(PR(t_1)/C(t_1) + ... + PR(t_m)/C(t_m))$, where d is the random jump factor, N is the total number of vertices in the graph, $\{t_i | 1 \leq i \leq m\}$ are the set of neighbors for the vertex v, and $PR(x)$ and $C(x)$ are the rank and the number of neighbor vertices of vertex x, respectively.

In the MapReduce-based implementation, the map takes a graph partition as input, and calculates the partial ranks for all the vertices within the partition. We use a hash table to maintain the partial rank of all the vertices. With the hash table, the map function scans the partition only once. During the scan, we need to fetch a vertex from the partition, and update the partial rank in the hash table. The $reduce$ aggregates the partial ranks on each vertex. The pseudo code of these two functions can be found in Appendix D.

3.2 Propagation

Iterative propagation is to transfer the information of each vertex to its neighbor vertices iteratively. At each iteration, the information transfer is occurred along the edges. This information flow consists of the basic pattern on traversing the graph in parallel.

Propagation supports two user-defined functions, namely $transfer$ and $combine$. 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 vertex/value pair 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: $(v, v') \rightarrow (v', value)$, where v' is v's neighbor.
combine: (v, 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 result; 2) the Combine stage: Surfer calls $combine$ on the intermediate results generated in the Transfer stage.

Propagation remedies the deficiency and the programmability issues in MapReduce. Propagation allows the underlying system to automatically exploit the locality of graph partitions, without laying the burden on developers. We discuss the tails on optimizations in Section 5. The Combine stage in propagation minimizes data shuffling along the cross-partition edges, unlike the costly data shuffling in the Reduce stage of MapReduce.

Implementing the network ranking example requires only a few code lines in the user-defined functions (see Appendix D). Compared with the user-defined functions with MapReduce, the propagation-based implementation is simple. With propagation, developers focus on their custom logic, as opposed to a mixture of graph access pattern and the application logic in MapReduce.

Propagation has the better programmability and the higher efficiency than MapReduce when the access pattern of the target application matches that of propagation, mainly edge-oriented tasks. However, when the access pattern of the tasks, e.g., vertex-oriented tasks, does not match propagation, it is tricky to implement the target

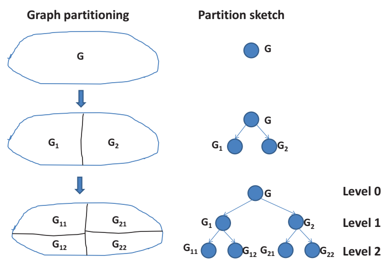


Figure 2: Graph partitioning and the partition sketch

application with propagation. One of the example application is to compute the histogram of the vertex degree. MapReduce has a simpler implementation than propagation. To emulate the process of MapReduce, Surfer introduces *virtual vertex* for developers to transfer the results to the designed virtual vertex, which virtually creates edges from the vertices in the graph to the virtual vertex. On the virtual vertex, Surfer performs combination on the values from the vertices in the graph.

4 Graph Partitioning in the Cloud

The way of partitioning and storing the graph is an important factor in the efficiency of Surfer. To address the network bandwidth unevenness in current cloud environment, we propose a bandwidth aware mechanism to improve the bandwidth utilization. The basic idea is to partition and store the graph partitions according to their numbers of cross-partition edges such that the partitions with a large number of cross-partition edges are stored in the machines with high network bandwidth. We start with modeling the graph partitioning process with multi-level bisections, and then develop our partitioning algorithm based on the model.

4.1 Graph Partitioning Model

We model the process of the multi-level graph partitioning algorithm as a binary tree (namely *partition sketch*). Each node in the partition sketch represents the graph as the input for a bisection in the entire graph partitioning process: the root node representing the input data graph; a non-leaf node at level $(i+1)$ representing the partitions of the i^{th} iteration; the leaf nodes representing the graph partitions generated by the multi-level graph partitioning algorithm.

The partition sketch is a balanced tree. If the number of graph partitions is P, the number of levels of the partition sketch is $(\log_2 P + 1)$. Figure 2 illustrates the correspondence between partition sketch and the bisections in the entire graph partitioning process. In the figure, the graph is divided into four partitions, and the partition sketch grows to three levels.

We further define an *ideal partition sketch* as a partition sketch via optimal bisections on each level. On each bisection, the optimal bisection minimizes the number of cross-partition edges between the two generated partitions. The ideal partition sketch represents the iterative partition process with the optimal bisection on each partition. This is the best case that existing bisection-based algorithms [18, 16, 14] can achieve. Partitioning with optimal bisections does not necessarily result in P partitions with globally minimum number of cross-partition edges. The bisection-based partitioning algorithms are simplified with the tradeoff between the algorithm complexity and the partitioning quality. Nevertheless, existing studies [16, 14] have demonstrated that they can achieve relatively good partitioning quality, approaching the global optimality. Thus, we use the ideal partition sketch to study the properties of the multi-level partitioning algorithm.

The ideal partition sketch has the following properties:

Local optimality. Denote $C(n_1, n_2)$ as the number of cross-partition edges between two nodes n_1 and n_2 in the partition sketch. Given any two nodes n_1 and n_2 with a common parent node p in the ideal partition sketch, we have $C(n_1, n_2)$ is the minimum among all the possible bisections on p.

By definition of the ideal partition sketch, the local optimality is achieved on each bisection.

Monotonicity. Suppose the total number of cross-partition edges among any partitions at the same level l in the partition sketch to be T_l. The monotonicity of the ideal partition sketch is that $T_i \leq T_j$, if $i \leq j$.

The monotonicity reflects the changes in the number of cross-partition edges in the recursive partitioning.

Proximity. Given any two nodes n_1 and n_2 with a common parent node p, any other two nodes n_3 and n_4 with a common parent node p', and p and p' are with the same parent, we have $C(n_1, n_2) + C(n_3, n_4) \geq C(n_{\pi(1)}, n_{\pi(2)}) + C(n_{\pi(3)}, n_{\pi(4)})$ where π is any permutation on (1, 2, 3, 4).

The proof of proximity can be found in Appendix C. The intuition of the proximity is, at a certain level of the ideal partition sketch, the partitions with a low common ancestor have a larger number of cross-partition edges than those with a high common ancestor.

These properties of the partitioning sketch indicate the following design principles for partitioning and storing graphs, in order to match the network bandwidth with the number of cross-partition edges.

P_1. Graph partitioning should gracefully adapt to the bandwidth unevenness in the cloud network. The number of cross-partition edges is a good indicator on bandwidth requirements. According to the local optimality, the two partitions generated in a bisection on a graph should be stored on two machine sets such that the aggregated bandwidth between the two machine sets is the lowest.

P_2. The partition size should be carefully chosen for the efficiency of processing. According to the monotonicity, a small partition size increases the number of levels of the partition sketch, resulting in a large number of cross-partition edges. On the other hand, a large partition may not fit into the main memory, which results in random disk I/O in accessing the graph data.

P_3. According to proximity, the nodes with a low common ancestor should be stored together in the machine sets with high interconnected bandwidth in order to reduce the performance impact of the large number of cross-partition edges.

4.2 Bandwidth Aware Graph Partitioning

With the three design principles in mind, we develop a unified algorithm for partitioning and storing the graph. The algorithm gracefully adapts the network bandwidth in the cloud to the number of cross-partition edges according to the partition sketch.

We model the machines for processing the data graph as a weighted graph (namely *machine graph*). In a machine graph, each vertex represents a machine. An edge means the connectivity between the two machines represented by the vertices associated with the edge, and the weight is the network bandwidth between them. We currently model the graph as an undirected graph, since the bandwidths are similar in both directions. Given a set of machines, the machine graph can be easily constructed by calibrating the network bandwidth between any two machines in the set. The left part of Figure 3 illustrates the machine graph for a four-machine cluster with a tree topology. The example cluster consists of two pods, and each pod consists of two machines. Assuming that the intra-pod network bandwidth is higher than the inter-pod one, and the intra-pod bandwidth is the same across pods, we have the machine graph with four vertices and six edges. The edge representing the intra-pod connection is thicker than that for the inter-pod connection, indicating the network bandwidth unevenness.

In the remainder of this paper, we refer "graph" as a graph, and "machine graph" and "data graph" as the machine graph constructed from a set of machines in the cloud, and the input graph for partitioning respectively.

In the bandwidth aware graph partitioning algorithm (details in Appendix E), Surfer simultaneously partitions the data graph and the machine graph with multi-level bisections. At a certain level, Surfer assigns the machines

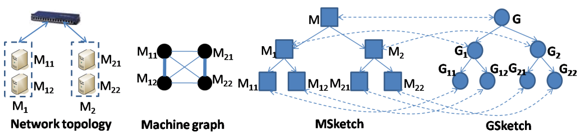


Figure 3: Mapping on the partition sketches between the machine graph and the data graph

in a partition of the machine graph to perform partitioning on the partition of the data graph. The suitable number of partitions, P, is determined in according to the available amount of main memory in the machine. This is because graph processing, especially for propagation, results in lots of random accesses on the partition. Thus, Surfer performs the computation on the graph partition when it is loaded into main memory. Denote the graph size to be $||G||$ bytes and the available memory in the machine to be r bytes, we calculate $P = 2^{\lceil \log_2 \frac{||G||}{r} \rceil}$. After $L = \lceil \log_2 \frac{||G||}{r} \rceil$ passes of bisection, each partition of the data graph can fit into the main memory for processing.

Surfer partitions the machine graph using a local graph partitioning algorithm such as Metis, since the number of machines is in the scale of tens of thousands, and the machine graph usually can fit into the main memory of a single machine. On the bisection of the machine graph, the objective function is to minimize the weight of the cross-partition edges with the constraint of two partitions having around the same number of machines. This objective function matches the bandwidth unevenness of the cloud. The goal of minimizing the weight of cross-partition edges in the machine graph corresponds to minimizing the number of cross-partition edges in the data graph. This is a graceful adaptation on assigning the network bandwidth to partitions with different number of cross-partition edges. The constraint of making partitions with the same number of machines is for load-balancing purpose, since partitions in the data graph also have similar sizes.

Along the multi-level bisections, the algorithm traverses the partition sketches of the machine graph and the data graph, and builds a mapping between the machine and the partition. The mapping guides the machines where the graph partition is further partitioned, and where the graph partition is stored in Surfer. Figure 3 demonstrates the mapping between an example machine graph and a data graph for the partitioning algorithm. The data graph partitioning is gracefully adapted to the bandwidth unevenness of the cluster. The bisection on the entire graph G is done on the entire cluster. At the next level, the bisections on G_1 and G_2 are performed on pods M_1 and M_2, respectively. Finally, the partitions are stored in the machines according to the mapping.

This mapping satisfies the three design principles on partitioning: 1) the number of cross-partition edges is gradually adapted to the network bandwidth. In each bisection of the recursion, the cut with the minimum number of cross-partition edges in the data graph coincides that with minimum aggregated bandwidth in the machine graph. 2) The partition size is tuned according to the amount of main memory available to reduce the random disk accesses. 3) In the recursion, the proximity among partitions in the machine graph matches that in the data graph.

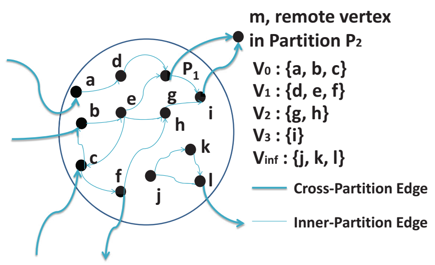


Figure 4: An example of cascaded execution of P-Surfer on a partition.

5 Optimizations in Propagation

Graph partitioning allows Surfer to exploit the locality of graph partitions. This enables a number of optimizations on reducing the network traffic. These optimization techniques are automatically applied during the runtime execution, without increasing the programming complexity on developers. Since the optimizations in MapReduce are similar to the previous study [5], this section focuses on the optimizations in our propagation-based Surfer (denoted as P-Surfer). We first present the execution on a single iteration of propagation in P-Surfer, followed by multiple iterations of propagation.

5.1 Single-Iteration Propagation

P-Surfer performs the propagation on the partitions in parallel on multiple machines. We briefly describe the basic flow, and present the details of the algorithm in Appendix E.

For each partition, P-Surfer has two stages, namely Transfer and Combine stage. In the Transfer stage, P-Surfer applies the *transfer* function on each edge with the optimizations. After the Transfer stage, all the intermediate results required for the Combine stage is stored on the same machine. If the source vertex is not in the current partition, its data is read along the cross-partition edge from the remote machine to the local machine. Along with each partition, P-Surfer stores the information about *boundary vertices* and *cross-partition edges* to capture the locality of the graph partition. The information is stored in two structures per partition: a hash table constructed from the set of boundary vertices (such as vertex a and c in Figure 4); a map on (v, pid), where v is the destination vertex of cross-partition edge, and pid is the ID of the remote partition that v belongs to. These two data structures are generated from graph partitioning, and are kept in the main memory when processing the corresponding partition.

In the Combine stage, P-Surfer applies the *combine* function on the intermediate result generated by the Transfer stage. Since the input data for the *combine* function are already in the local machine, there is no network traffic in this stage. Finally, the results are written to the disk.

The data transfer among graph partitions in P-Surfer is developed with the low-level communication primitives. The complexity of such low-level communication is hidden from developers.

Within a partition, P-Surfer automatically applies local propagation and local combination to reduce the network traffic.

Local propagation. Local propagation is to perform the propagation logic on an *inner vertex* within a partition. It can be performed on an inner vertex, because all its neighbors are stored within the same partition. The local propagation is performed on the intermediate results generated from the `transfer` stage within the partition. For example, vertices d, e, g and h are feasible for local propagation in Figure 4. The performance improvement by the local propagation is determined by the inner vertex ratio. The higher the inner vertex ratio, the more vertices the local propagation can be applied to.

Local combination. Similar to MapReduce, P-Surfer supports local combination when the `combine` function is annotated as an associative function. P-Surfer applies the local combination on the boundary vertices belonging to the same remote partition, and sends the combined intermediate results back to the local partition for the Combine stage. We consider local combination on boundary vertices only, since local propagation has been applied on the inner vertex. For example, in Figure 4, the local combination can be performed on vertices g and i, and only their combined value is sent to common remote vertex m. Local combination is similar to local propagation in reducing the size of intermediate results. The difference is that, local combination requires the associativity of the `combine` function, whereas local propagation does not. In contrast, local propagation has requirement on the graph structure within a partition, whereas local combination does not.

5.2 Multi-Iteration Propagation

Some applications such as PageRank [19] require running propagation with multiple iterations. A naive approach is to perform the single-iteration propagation multiple times. However, this approach results in significant disk I/O, each iteration reading the result of the previous iteration from the disk, and writing the result of the current iteration to the disk. There are opportunities for reducing the disk I/O. They are based on the observation: given a vertex v in the partition p, if all the k hop connected vertex for v are also in p, we can perform k iterations of propagation on v with a scan on p. We define the set of such vertices in the partition p as V_k. For any vertex v in V_k, the input values required in the k iterations of propagation are available, and thus we can perform the k iterations in a batch. We call this pattern *cascaded propagation*. We denote the vertices in a cycle to V_{inf}, since their neighbors at an arbitrary number of hops is also within the partition.

With the definition of V_k, we can also make an observation: except V_{inf}, the maximum k value in a partition is the diameter of the partition. For simplicity, we set the suitable number of iterations in a cascaded propagation to be the smallest diameter of all the partitions, d_{min}.

P-Surfer with cascaded propagation with I iterations is performed in two steps. The algorithm first processes V_{inf}, since they are not limited by the number of iterations. Next, the algorithm divides the I iterations into multiple phases. Each phase performs d_{min} iterations with cascaded propagation.

The amount of disk I/O saving depends on the ratio of V_k ($k \geq 2$). Figure 4 illustrates V_0, ..., V_3, and V_{inf} for a partition. V_0 contains all the boundary vertices in the partition. If an application requires three iterations of propagation, we can compute the final result at once in first iteration for vertices in V_3.

6 EVALUATION

In this section, we present the experimental results on Surfer with real-world and synthetic graphs.

6.1 Experimental Setup

We have implemented and evaluated Surfer on a real deployment of a cluster with 32 nodes with a Quad Intel Xeon CPU and 8 GB RAM each. All the machines form a pod, sharing the same switch. The current cluster provides even network bandwidth between any two machines. We denote the setting of the current cluster to be T_1.

We simulate different network environments in the cloud. In particular, we use software techniques to simulate the impact of different network topologies and hardware configurations. The basic idea is to add the latency to the network transfer according to the bandwidth of the connection between two machines. We consider the following two settings T_2 and T_3.

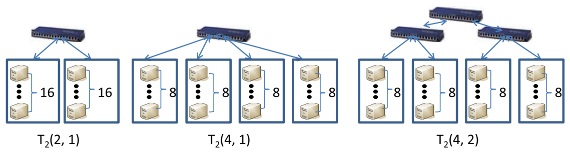


Figure 5: The variants of topology T_2.

Table 1: Elapsed time of partitioning on different topologies (hours)

Topology	T_1	T_2(2,1)	T_2(4,1)	T_2(4,2)	T_3
ParMetis	27.1	67.6	87.6	131.0	108.0
Bandwidth aware	27.1	33.8	43.9	58.3	64.9

T_2 simulates the popular tree network topology in the cloud [7]. We use $< \#pod, \#level >$ to represent the configuration of the tree topology, where $\#pod$ is the number of pods used for graph processing, and $\#level$ is the number of levels in the topology. Figure 5 shows the three variants of T_2 with 32 machines examined in our experiments. By default, the machine-machine bandwidth for T_2 is set to be 1/32 on the switch at the top level, and 1/16 on the switch at the second level.

T_3 simulates a cluster of different hardware configurations, with one half machines with lower bandwidth than the other half. We reduce the bandwidth from the machine with low bandwidth by one half.

The workloads consist of common graph applications described in Appendix D. The applications include network ranking (*NR*), recommender system (*RS*), triangle counting (*TC*), vertex degree distribution (*VDD*), reverse link graph (*RLG*), and calculating two-hop friend list (*TFL*).

The data sets include a snapshot of the MSN social network in 2007 and synthetic graphs, each of which is over 100GB. The MSN network used in this study contains 508.7 millions vertices and 29.6 billion edges. The number of edges in the MSN graph is almost five times as many as the largest one in the previous study [13]. Our evaluation is mainly on the real-world social network, and the synthetic graphs are used for parametric studies.

More details on our experimental setup and results including those on scalability and fault-tolerance of Surfer can be found in Appendix F.

6.2 Results on Partitioning

Table 1 shows the elapsed time for partitioning with our bandwidth aware graph partitioning algorithm, and with ParMetis [18] on T_1, T_2, and T_3. The number of partitions is 64, and each partition is around 2GB. Such a partition size achieves a good partitioning quality, and the processing on the partition can fit into the main memory.

On different environments (except T_1), the bandwidth aware graph partitioning algorithm achieves an improvement of 39–55% over ParMetis. ParMetis randomly chooses the available machine for processing, which is unaware of the network bandwidth unevenness. In contrast, by adapting the graph partitioning to the network bandwidth, our algorithm effectively utilizes the network bandwidth, and reduces the elapsed time of partitioning. This demonstrates the importance of the three design principles of an efficient graph partitioning algorithm in the cloud. Note that both techniques on T_1 behave the same, since every machine pair in T_1 has the same network bandwidth.

6.3 Results on Propagation

Single-iteration propagation. We first study the impact of optimization techniques in the execution in a single-iteration propagation. Tables 2 and 3 show the timing and I/O metrics on the applications on T_1. We implement these applications with Surfer in the following optimization levels.

O1. Surfer with graph partition storage layout generated from ParMetis, and no other optimizations.

O2. Surfer with storage according to the machine graph partitioning sketch, and no other optimizations.

O3. Surfer with local combination and local propagation, but with graph partition storage layout generated from ParMetis.

O4. Surfer with local combination and local propagation, with storage according to machine graph partitioning sketch.

The difference between O1 and O2 as well as between O3 and O4 is the comparison between the ParMetis' layout and our bandwidth aware algorithm. The difference between O1 and O3 as well as between O2 and O4 is to evaluate the effectiveness of our local optimizations (i.e., local combination and local propagation) in Surfer. Overall, we found O4 optimizes the performance dramatically. We make the following observations on the optimizations on the topology T_1.

First, comparing O1 with O2 and O3 with O4, we observed that the bandwidth aware graph partitioning improves the overall performance. Without local combination or local propagation, the performance improvement is 3–17%. When both techniques are enabled, the performance improvement is better, between 6% and 29%. On T_1, the performance improvement is contributed from the intra-machine locality, since partitions with common ancestor nodes in the partition sketch are stored on the

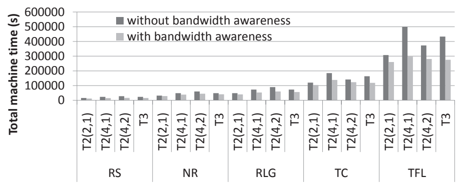


Figure 6: Impact of bandwidth aware partitioning on different topologies

same machine. Surfer schedules the execution according to the partition sketch and takes advantage of such locality. For propagation-emulated VDD, bandwidth ware graph partitioning has little improvement, since VDD is a vertex-oriented task.

Second, comparing O1 with O3 and O2 with O4, local optimization techniques significantly reduces the network I/O and disk I/O, and contributes to the overall performance improvement. In specific, the performance improvement of these two optimization techniques is 22–86% on the ParMetis' layout, and 23%–87% on the storage according to the machine graph partitioning sketch.

Therefore, comparing O1 with O4, the storage layout and the two local optimizations are accumulative. Their combined performance improvement is between 36% and 88%. Among the applications in our benchmark, the performance improvement for NR and TFL is relatively high. Because these two applications generate huge amounts of intermediate data, and local optimizations significantly reduces the data transfer, especially when the data layout is according to the bandwidth aware algorithm.

We further investigate the impact of bandwidth aware graph partitioning algorithm on different network environments. Figure 6 shows the optimized propagation with and without awareness of bandwidth on T_2 and T_3. Bandwidth aware graph partitioning significantly improves the performance on different network topologies, with an improvement up to 71%.

Multi-Iteration Propagation. The performance improvement of cascaded propagation highly depends on the structure of the graph. We studied the sets of V_k used in the cascaded propagation on the MSN network, and found that the ratio of vertices in V_k ($k \geq 2$) is 7%. That means, only 7% of the vertices can gain the benefit of cascaded propagation.

We evaluate the cascaded optimization on the network ranking. In the experiment, we run the network ranking in different numbers of iterations. The results show that cascaded propagation further improves the performance of the optimized propagation with multiple iterations. When the number of iteration is three, cascaded propagation improves response time by 8% and reduces total disk IO by 12%. The performance improvement is

Table 2: Response time and total machine time of applications on T_1 (Seconds)

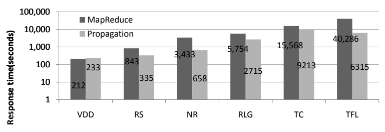
	VDD		RS		NR		RLG		TC		TFL	
	Res.	Total.	Res.	Total.	Res.	Total.	Res.	Total.	Res.	Total.	Res.	Total.
O1	325	5523	592	9291	3421	48498	3815	47213	20125	156243	43245	607854
O2	325	5523	436	8954	2653	46823	3124	39212	17431	134091	38212	589967
O3	233	3658	518	7220	736	14278	2994	32140	5426	98645	77345	82529
O4	**233**	**3658**	**273**	**5133**	**658**	**12872**	**2715**	**30173**	**3335**	**85568**	**6315**	**75657**

Table 3: Disk and network I/O of applications on T_1 (GB)

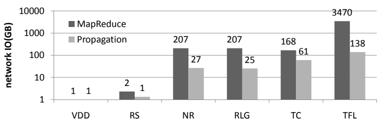
	VDD		RS		NR		RLG		TC		TFL	
	Network.	Disk.	Network.	Disk.	Network.	Disk.	Network.	Disk.	Network.	Disk.	Network.	Disk.
O1	3	127	13	162	136	619	93	553	87	1325	2886	7087
O2	3	127	11	160	114	570	58	477	72	1202	2271	4908
O3	1	122	5	133	28	183	28	303	65	265	169	651
O4	**1**	**122**	**5**	**132**	**27**	**181**	**25**	**263**	**61**	**255**	**138**	**618**

Table 4: Number of source code lines in user-defined functions

Engine	*VDD*	*NR*	*RS*	*RLG*	*TC*	*TFL*
Hadoop	24	147	152	131	157	171
Home-grown MapReduce	33	163	168	144	171	194
Propagation	18	21	22	23	27	25



(a) Response Time



(b) Network Traffic

Figure 7: Performance comparison between MapReduce and P-Surfer

stable as we increase the number of iterations. The performance improvement matches our observation on the ratio of V_k ($k \geq 2$).

6.4 Propagation vs. MapReduce

Table 4 shows the number of source code lines in user-defined functions in Hadoop, our home-grown MapReduce and Surfer. Our home-grown MapReduce has a similar code size to Hadoop. The slight difference is for ours is implemented in C++ and that of Hadoop is in Java. Compared with both MapReduce variants, propagation does not have to handle the details on graph partitions, and thus have a much smaller code size. This demonstrates a good programmability on propagation.

Figure 7 shows the performance comparison between MapReduce and propagation on the applications. Propagation is significantly faster than MapReduce on most applications (except VDD). In particular, the speedup on the response time is between 1.7 to 5.8 times. The major contributor of the speedup is the reduction in network IO: MapReduce with a data shuffling in the Reduce stage, whereas propagation incurring network traffic only for cross-partition edges. Specifically, propagation-based implementations have 42.3–96.0% less network I/O than their MapReduce-based counterparts. Emulating MapReduce in VDD, propagation has a similar performance on MapReduce.

7 Conclusion

As large graph processing becomes popular in the cloud, we develop Surfer towards a full-fledged large graph processing system that scales in a large number of machines in the cloud. In this paper, we aim at improving its efficiency and programmability, through supporting two optimized primitives MapReduce and propagation on partitioned graph. Adapting the unique network environment in the cloud, we develop a bandwidth aware graph partitioning algorithm to minimize the network traffic in partitioning and processing. Our evaluation on the large real-world and synthetic graphs (over 100GB each) shows that 1) the bandwidth aware graph partitioning improves partitioning by 39–55%, and the execution of propagation by 6–28%; 2) propagation is 1.7 to 5.8 times faster than MapReduce, and with fewer source code lines by developers.

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] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Fourth SIAM International Conference on Data Mining*, April 2004.

[3] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *SIGMOD (demo)*, 2010.

[4] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *ACM SIGMOD*, 2007.

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

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[7] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. *SIGCOMM*, 38(4), 2008.

[8] Hadoop. *http://hadoop.apache.org/*.

[9] T. Hogg and L. Adamic. Enhancing reputation mechanisms via online social networks. In *ACM conference on Electronic commerce*, 2004.

[10] 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.

[11] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[12] 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.

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

[14] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing*, 1996.

[15] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[16] 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.

[17] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[18] Metis. *http://glaros.dtc.umn.edu/gkhome/views/metis/*.

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

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

[21] 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.

[22] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. 2003.

[23] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[25] A. Zhou, W. Qian, D. Tao, and Q. Ma. DisG: A distributed graph repository for web infrastructure (invited paper). *International Symposium on Universal Communication*, 0:141–145, 2008.

A More Related Work

A.1 Data-Intensive Computing Systems

The increasingly large number of machines in the cloud raises a number of cloud systems including MapReduce [5], Hadoop [8] and Dryad [10]. There have been multiple variants or extensions, such as Map-Reduce-Merge [4] and DryadLINQ [23]. All of these systems allow the data analysts to easily write programs to manipulate their large scale data sets without worrying about the complexity of distributed systems.

Let us briefly review the software stack of the current MapReduce implementation [5]. MapReduce provides two APIs for the developer to implement: `map` and `reduce`. The `map` function takes an input key/value pair (k_1, v_1) and outputs a list of intermediate key/value pairs (k_2, v_2). The `reduce` function takes all values associated with the same key and produces a list of key/value pairs. Developers implement the application logic inside `map` and `reduce`. The MapReduce runtime manages the parallel execution of these two functions. The runtime consists of three major stages: (1) the Map stage processes a unit of the input file and outputs (key, value) pairs; (2) the Shuffling stage: MapReduce automatically groups of all values by key using hash partitioning, (3) the Reduce stage processes the values with the same key and outputs the final result. The data is stored in the distributed and replicated file system named GFS [6].

A.2 Graph Bisection

Since graph bisection is a key operation in parallel graph partitioning [16, 14], we briefly introduce the process of bisection.

Figure 8 illustrates the three phases in a graph bisection, namely *coarsening*, *partitioning* and *uncoarsening*. The coarsening phase consists of multiple iterations. In each iteration, multiple vertices in the graph are coarsened into one according to some heuristics, and the graph is condensed into a smaller graph. The coarsening phase ends when the graph is small enough, in the scale of thousands of vertices. The partitioning phase divides the coarsened graph into two partitions using a sequential and high-quality partitioning algorithm such as GGGP (Greedy Graph Growing Partitioning) [15]. In the uncoarsening phase, the partitions are then iteratively projected back towards the original graph, with a local refinement on each iteration. The solid line in Figure 8 represents the cut after uncoarsening, and the dotted line represents the cut after refinement. Local refinement can significantly improve the partition quality [16]. The iterations are highly parallelizable, and their scalability has been evaluated on shared-memory architectures (such as Cray supercomputers) [16, 14].

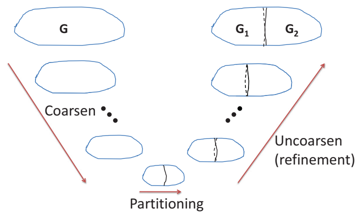


Figure 8: The three phases in a graph bisection

B Architecture of Surfer

Given a graph processing job, Surfer performs the execution in two steps.
Step 1. A code generator in Surfer takes the user-defined functions as input, and generates the executable for a distributed execution.
Step 2. Surfer performs the execution on the slave nodes. During the execution, the job scheduler selects a machine as the job manager, and the job manager dispatches the tasks to the slave nodes. The job manager also records resource utilization and estimates the execution progress of the job. Surfer provides a GUI (Graphical User Interface) [3] to facilitate the programming in Step 1, and displays the runtime dynamics to facilitate developers to debug and analyze Step 2.

In the current system, it is developers' choice to select either MapReduce or propagation to implement their tasks. As shown in our experimental evaluation, propagation-based implementations are mostly more efficient than their MapReduce-based counterparts. Thus, developers may choose propagation for the applications matching the access pattern of propagation, and MapReduce for other tasks. We are developing a high-level language on top of MapReduce and propagation, to further improve the programmability of Surfer.

The current job manager in Surfer is simple. It dispatches one more task to a slave node, when the slave node finishes a task. Additionally, it provides executions with fault tolerance to machine and task failures. The job manager detects the failed slave machine using heartbeat messages, and automatically handles the machine or task failures. A machine failure can result in multiple task failures. Surfer handles the machine failure according to the task type. If the task is a Transfer task, Surfer simply puts the task into the waiting list and re-executes it. If it is a Combine task, the recovery requires re-transferring the input from remote machines according to incoming edges, prior to the re-execution. The data re-transmission requires the knowledge on which partition the incoming edge is from. Instead of maintaining a global mapping from an arbitrary vertex ID to its partition ID, we encode the vertex IDs such that the vertex IDs within a partition compose a consecutive range. Suppose a vertex is the jth vertex in partition i ($0 \leq i \leq (P-1)$), the encoded ID of the vertex is $\sum_{k=0}^{P-1} p_k + j$, where p_k is the number of vertices in partition k. Surfer maintains the number of vertices in each partition. From this encoding, it is straightforward to find the partition ID for a vertex.

C Proof of proximity in graph partition model

Proximity. Given any two nodes n_1 and n_2 with a common parent node p, any other two nodes n_3 and n_4 with a common parent node p', and p and p' are with the same parent, we have $C(n_1, n_2) + C(n_3, n_4) \geq C(n_{\pi(1)}, n_{\pi(2)}) + C(n_{\pi(3)}, n_{\pi(4)})$ where π is any permutation on (1, 2, 3, 4).

Proof. According to local optimality, we know that $C(p, p') = C(n_1, n_3) + C(n_1, n_4) + C(n_2, n_3) + C(n_2, n_4)$ is the minimum. Thus, we have:

$$C(n_1, n_2) + C(n_1, n_4) + C(n_3, n_2) + C(n_3, n_4) \geq C(p, p') \tag{1}$$
$$C(n_1, n_2) + C(n_1, n_3) + C(n_4, n_2) + C(n_4, n_3) \geq C(p, p') \tag{2}$$

Substituting $C(p, p')$, we have

$$C(n_1, n_3) + C(n_2, n_4) \leq C(n_1, n_2) + C(n_3, n_4) \tag{3}$$
$$C(n_2, n_3) + C(n_1, n_4) \leq C(n_1, n_2) + C(n_3, n_4) \tag{4}$$

That means, we have $C(n_1, n_2) + C(n_3, n_4) \geq C(n_{\pi(1)}, n_{\pi(2)}) + C(n_{\pi(3)}, n_{\pi(4)})$ where π is any permutation on (1, 2, 3, 4).

D Applications

In this section, we present the implementation of several common applications in social network using propagation. These applications represent basic processing on graphs. We can find their counterparts on other graph applications such as web graph analysis.

Network ranking (*NR*). Network ranking is to generate a ranking on the vertices in the graph using PageRank [19] or its variants. Applying ranking to social network is used in assigning reputation to individuals and finding the influential persons in the social network [9]. Algorithm 1 illustrates the user-defined functions of network ranking with propagation.

Algorithm 1 User-defined procedures in propagation-based PageRank

```
Procedure: Transfer ($v$, $v'$)
  1: $Emit$ ($v'$, $\frac{v.rank \times d}{|v.neighbor|}$);
Procedure: Combine ($v$, $valueList$)
  1: for each value $r$ in $valueList$ do
  2:     $v.rank$+=$r$;
  3: $v.rank$+=(1-$d$)/$N$;
  4: $Emit$ ($v$);
```

As an example of comparing MapReduce and propagation, we introduce the user-defined functions in the MapReduce-based implementation. Algorithm 2 illustrates user-defined functions of network ranking with MapReduce. We use a hash table to maintain the partial rank of all the vertices. With the hash table, the *map* function scans the partition only once. During the scan, we need to fetch a vertex from the partition, and update the partial rank in the hash table.

Algorithm 2 User-defined procedures in MapReduce- and propagation-based PageRank

```
//MapReduce-based implementation;
Procedure: map ($p$)
Description: Compute the partial ranks in Partition $p$, and maintain them in a hash table $rTable$.
1: for each vertex $v$ in $p$ do
2:     $\delta = \frac{v.rank \times d}{|v.neighbor|}$;
3:     for Vertex $v'$ in $v.neighbors$ do
4:         if $v'$ is not in $rTable$ then
5:             $rTable$.Add($v'$, $\delta$);
6:         else
7:             $rTable[v']$+= $\delta$;
8: for each vertex $v$ in $rTable$ do
9:     $Emit$ ($v$, $rTable[v]$);
Procedure: reduce ($v$, $valueList$)
1: for each value $r$ in $valueList$ do
2:     $v.rank$+=$r$;
3: $v.rank$+=(1-$d$)/$N$;
4: $Emit$ ($v$, $v.rank$);
```

Recommender system (*RS*). A simple recommender system based on social network [1]. For example, the system can show the advertisement of the product one uses or the pages one views to his/her friends in the network. We simulate a product recommending process in a social network. In particular, we investigate how the advertisement of a certain product propagates in the network. The recommending starts with a set of initial vertices who have used the product. For each individual using the product in the network, i.e., the $useProduct$ value of the individual is true, the recommender system recommends the product to all his/her friends. Each person can accept the product recommending with a probability p. After a few iterations of propagation, the recommender system can examine the effectiveness of the recommending.

Triangle counting (*TC*). Triangle counting has its applications in web spam detection and social network. Previous studies [21] show that the amount of triangles in the social network of a user is a good indicator of the role of that user in the community.

A triangle in the graph is defined as three vertices, where there is an edge connected any two vertices among them. Triangle counting requires a single iteration for propagation on the graph. In the experiment, we pick the subgraph from selecting a subset of vertices from the large graph. Algorithm 3 shows the user-defined procedures in propagation-based triangle counting. In the Transfer stage, we first select this subset of vertices in the graph. In our experiments, the ratio of selected vertices is 10%. Next, we transfer the neighbor list of the source vertex of every edge to the target vertex. In the Combine stage, we check whether the adjacent list has overlapping with any of the awarded neighbor lists. A procedure $checkOverlapping$ is used to check whether we find a triangle. But, we may count the same triangle three times. To remove the duplicate counting, we can assume an order of the vertices, say in the alphabetic order. We only emit the vertex with the minimum ID.

Algorithm 3 User-defined procedures in Surfer-based triangle counting

```
Procedure: Transfer ($v$, $v'$)
Description: Transfer the neighbor list along the edge between $v$ and $v'$.
1: $Emit$ ($v'$, $v$'s neighbor list);
Procedure: Combine ($v$, $valueList$)
Description: Output the triangle.
1: for each list $l$ in $valueList$ do
2:     if $checkOverlapping$ ($v$'s neighbor list, $l$) is true then
3:         $Emit$ ($v'$, 1);
```

Vertex Degree Distribution (*VDD*). The degree distribution is an important property of a social graph. For example, many social networks have demonstrated the power-law distribution on vertex degrees. VDD is a vertex-oriented task, where we use *virtual vertices* and *virtual edges* for implementation. In the Transfer stage, we emit the degree in the adjacency list to the target virtual vertex. The virtual vertex ID is the same as the value of the degree. In the Combine stage, only the virtual vertex performs the combination on the data generated in the Transfer stage.

Reverse Link Graph (*RLG*). *RLG* is to process all the incoming edges for the directed graph. The task is to reverse the source vertex and destination vertex for each edge in the graph, and to store the reversed graph as adjacency list among machines. In the Transfer stage, we transfer the reversed edge to the new destination vertex. In the Combine stage, we count the number of neighbors in the reversed links, and store the neighbors in an adjacency list. Then we emit the new graph out.

Two-hop Friends List (*TFL*). The operation of finding the list of two-hop friends has been used for analyzing social influence spread, community detection and so on. We use push operation to aggregate two-hop friends at each vertex. In Transfer stage, we select a subset of vertices in the graph. The ratio of selected vertices is 10% in our experiments. Every selected vertex pushes its neighbor list to each of its neighbors. Thus, it receives the neighbor list of its neighbors. In Combine stage, we store distinct vertices in received neighbor lists as the adjacency list of the destination vertex.

E Detailed Algorithms

Algorithm 4 illustrates the bandwidth aware graph partitioning algorithm in Surfer.

Algorithm 4 Bandwidth aware graph partitioning in Surfer

Input: A set of machines S in the cloud, the data graph G, the number of partitions P ($P = 2^L$, $L = \lceil \log_2 \frac{||G||}{r} \rceil$)
Description: Partition G into P partitions with S

```
1: Construct the machine graph $M$ from $S$;
2: BAPart($M$, $G$, 1);//the first level of recursive calls.
```

Procedures: BAPart(M, G, l)

```
1:  Divide $G$ into two partitions ($G_1$ and $G_2$) with the machines in $M$;
2:  if $M$ consists of a single machine then
3:      Let the machine in $M$ be $m$.
4:      Divide $G$ into $2^{L-l}$ partitions using $m$ with the local partitioning algorithm;
5:      Store the result partitions in $m$;
6:  else
7:      if $l = L$ then
8:          Select the machine with the maximum aggregated bandwidth in $M$, and let it be $m$;
9:          Store $G$ in $m$;
10:     else
11:         Divide $M$ into two partitions $M_1$ and $M_2$;
12:         Divide $G$ into two partitions $G_1$ and $G_2$ with the machines in $M$;
13:         BAPart($M_1$, $G_1$, $l$+1);
14:         BAPart($M_2$, $G_2$, $l$+1);
```

The detailed process of single-iteration propagation is illustrated in Algorithm 5.

F More Details on Experiments

F.1 Experimental Setup

The cluster consists of 32 machines, each with a Quad Intel Xeon X3360 running at 2.83GHz, 8 GB memory and two 1TB SATA disks, connected with 1 Gb Ethernet. The operating system is Windows Server 2003.

We implement Surfer in C++, compiled in Visual Studio 9 with full optimizations enabled. We implement our home-grown MapReduce primitive, following the design and implementation described by Google [5]. We do not provide the performance numbers on Hadoop, since Hadoop is not supported as a production platform in Windows [8]. For a fair comparison on the programmability with Hadoop, we have also implemented all the applications with Java on Hadoop.

We use two metrics for the time efficiency: the response time and the total machine time, where the response time is the elapsed time from submitting the task to Surfer till its completion, and the total machine time is the aggregated total amount of time spent on the entire task on all the machines involved. To understand the effectiveness of our optimizations, we report two I/O metrics: the total network I/O and the total disk I/O during

Algorithm 5 Basic flow of a single iteration of propagation

Input: Graph $G(V, E)$
Description: Perform one iteration of graph propagation with `transfer` and `combine` functions.

```
1:  for each partition $p$ of $G$ do
2:      // The Transfer stage: the following loop is performed in parallel.
3:      Read $p$ and associated values from disk;
4:      for each non-cross-partition edge $e$ in $p$ do
5:          call transfer on $e$;
6:      for each vertex $v$ in $p$ do
7:          if $v$ is an inner vertex of partition $p$ then
8:              perform local propagation optimization;
9:      //Handling the cross-partition edges.
10:     if combine is associative then
11:         perform local grouping optimization on $pid$ in the map $(v, pid)$;
12:         for each group $g$ in the result of local grouping do
13:             call transfer on $e$, where $e$ is a cross-partition edge in $g$;
14:             read the intermediate results from $g$ with local combination remotely;
15:     else
16:         for each cross-partition edge $e$ in $p$ do
17:             read the vertex that is not in the current partition remotely;
18:             call transfer on $e$;
19: //The Combine stage: the loop is performed in parallel. After all the incoming data is transferred to local machine.
20: for each vertex $v$ in $p$ do
21:     call combine on $v$ and its value list;
22: Write updated $p$ and values to disk;
```

Table 5: The statistics of inner edge ratios with different partition sizes

Number of partitions	128	64	32	16
Partition granularity (GB)	1	2	4	8
ier of our partitioning(%)	50.3	57.7	65.5	72.7
ier of random partitioning(%)	1.4	2.2	4.1	6.8

the execution. We ran each experiment three times, and report the average value for each metric. The measured metric values are stable across different runs.

Simulating Different Network environments. T_2 is a tree topology. At each level of the tree topology, we estimate the average bandwidth given to each machine pair on all-to-all communication, since all-to-all communication is common in the data shuffling [7]. Suppose the bandwidth of the switch is B_{switch} Gb/sec. If all the machines perform all-to-all communication, the worst-case bandwidth for any pair of two machines sharing the switch is $B_{switch}/(\prod_{i=1}^{\#pod} N_i \times (N - N_i))$, where N_i is the number of machines in the ith pod, and N is the total number of machines in the cluster.

Our software approaches simulate the cross-pod bandwidth by add a latency into the send/receive operations such that the time for sending the data matches a specific bandwidth. Suppose we will send N bits. If the target machine is in the same pod, we use send/receive operations as the normal ones. Otherwise, we trigger send/receive with a delay of $\frac{N}{B_{switch}/(\prod_{i=1}^{\#pod} N_i \times (N-N_i))}$ seconds such that the elapsed time for finishing the operation matches the tree topology in the worst case. In our experiments, the default delay is set to be 32 times on the switch at the top level, and 16 times on the switch at the second level.

T_3 simulate a pod whose machines have two different configurations. For simplicity, we simulate one half of the machines randomly chosen from the pod having one half bandwidth of the remainder machines in the cluster. Let the former set of machines be LOW and the rest machines belonging to set $HIGH$. If both of the source and the target machines is in $HIGH$, we use send/receive operations as the normal ones. Otherwise, we add a delay the same as the data transfer time, since the network bandwidth between two machines is limited by the smaller one.

Synthetic graph generation. We generate synthetic graphs simulating small world phenomenon. We first generate multiple small graphs with small-world characteristics using an existing generator [2], and next randomly change a ratio (p_r) of edges to connect these small graphs into a large graph. The default value of p_r is 5%. We varied the sizes of the synthetic graphs for evaluating the scalability of Surfer. The default size is 100GB, with 408.4 million vertices and 25.9 billion edges.

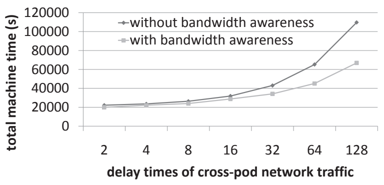


Figure 9: The impact of time delay parameters for cross pod traffic in NR

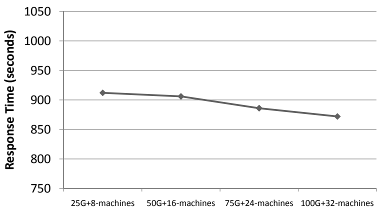


Figure 11: Response time of P-Surfer on different cluster configurations and synthetic graphs

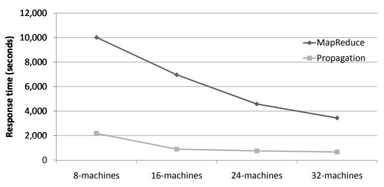
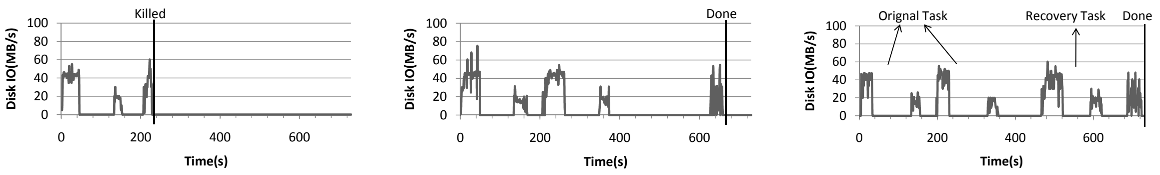


Figure 12: Comparison of network ranking between MapReduce and P-Surfer



(a) Killed task (b) Normal execution (c) Recovery on another machine

Figure 10: Disk I/O rates rates over time for different executions of the NR

F.2 Experimental results

Partitioning quality. We first investigate how the partition size affects the quality of partitioning. We quantify the partitioning quality with the inner edge ratio, $ier = \frac{ie}{|E|}$, where ie and $|E|$ are the number of inner edges and the total number of edges in the graph. Table 5 shows the ier values and the partition granularity with the number of partitions varied. As we vary the number of partitions from 16 to 128, the partition size decreases from 8GB to 1GB, and the ier ratio decreases from 72.7% to 50.3%. This validates the monotonicity of graph partitioning: as the number of levels of the partition sketch increases, the number of cross-partition edges increases. Although the partition size at 4GB or 8GB provides higher inner edge ratios, the partition and the intermediate data generated from Surfer usually cannot fit into the main memory, and cause a huge amount of random disk I/Os. Therefore, we choose 2GB as our default setting, and divide the MSN graph into 64 partitions.

As a sanity check on the partition granularity, we show the partition quality of randomly partitioning the graph. The ier ratios of our partitioning algorithm are significantly higher than those of random partitioning. This confirms that graph partitioning achieves a high inner edge ratio.

Impact of cross-pod network delay. We investigate the impact of the simulated cross-pod network delays. Figure 9 shows the results of network ranking on $T_2(2,1)$ when the simulated delay is varied from twice to 128 times. As the simulated delay increases, the performance improvement of the bandwidth aware graph partitioning algorithm becomes more significant. That means, the bandwidth aware algorithm is very helpful when the scale of the data center is huge.

Fault tolerance. Figure 10 shows the disk I/O rates of an execution of the network ranking, where we intentionally kill a slave node at 235 seconds (as shown in Figure 10(a)). Upon detecting the failure, the job manager immediately puts the task into the waiting list, and later restarts the task on another available slave node. The re-execution happens in another slave node (shown in Figure 10(c)). Comparing the normal execution in Figure 10(b), the entire computation with recovery finishes in 723 seconds including a startup overhead of 10% over the normal execution.

Scalability. We evaluate scalability of P-Surfer for network ranking through increasing the number of machines and meanwhile increasing the size of synthetic graphs. Figure 11 shows the response time of P-Surfer when the number of machines is varied from 8 to 32. As the number of machines and graph size increase, the response time slightly decreases, indicating that P-Surfer has good scalability to accommodate increasing loads on increasing number of machines.

MapReduce vs. propagation. Figure 12 shows the performance comparison between MapReduce and propagation for network ranking with the number of machines varied from 8, 16, 24, and 32. Propagation is 4.6 to 7.8 times faster than MapReduce.