Improving Large Graph Processing on Partitioned Graphs in the Cloud

Rishan Chen
Shenzhen Graduate School of Peking University, China
crs@net.pku.edu.cn

Xuetian Weng
Shenzhen Graduate School of Peking University, China
wengxt@gmail.com

Bingsheng He
Nanyang Technological University
bshe@ntu.edu.sg

Mao Yang
Microsoft Research Asia
maoyang@microsoft.com

Byron Choi
Hong Kong Baptist University
bchoi@comp.hkbu.edu.hk

Xiaoming Li
Shenzhen Graduate School of Peking University, China
lxm@pku.edu.cn

ABSTRACT

As the study of large graphs over hundreds of gigabytes becomes increasingly popular for various data-intensive applications in cloud computing, developing large graph processing systems has become a hot and fruitful research area. Many of those existing systems support a *vertex-oriented* execution model and allow users to develop custom logics on vertices. However, the inherently random access pattern on the vertex-oriented computation generates a significant amount of network traffic. While graph partitioning is known to be effective to reduce network traffic in graph processing, there is little attention given to how graph partitioning can be effectively integrated into large graph processing in the cloud environment. In this paper, we develop a novel graph partitioning framework to improve the network performance of graph partitioning itself, partitioned graph storage and vertex-oriented graph processing. All optimizations are specifically designed for the cloud network environment. In experiments, we develop a system prototype following Pregel (the latest vertex-oriented graph engine by Google), and extend it with our graph partitioning framework. We conduct the experiments with a real-world social network and synthetic graphs over 100GB each in a local cluster and on Amazon EC2. Our experimental results demonstrate the efficiency of our graph partitioning framework, and the effectiveness of network performance aware optimizations on the large graph processing engine.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special purpose and Application based Systems; C.4.1 [**Computer Systems Organization**]: Performance of Systems—*design studies*

General Terms

Design, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOCC'12, October 14-17, 2012, San Jose, CA USA
Copyright 2012 ACM 978-1-4503-1761-0/12/10 ...$15.00.

Keywords

Large Graph Processing, Graph Partitioning, Cloud Computing, Data Center Network

1. INTRODUCTION

Large graph processing has become popular for various data-intensive applications on increasingly large web and social networks [20, 21]. Due to the ever increasing size of graphs, application deployments are moving from a small number of HPC servers or super computers [24, 13] towards the cloud with a large number of commodity servers [29, 21]. Most processing tasks in these graph applications mainly involve batch operations in which many vertices and/or edges of the graph are accessed. Examples of these tasks include PageRank [31], reverse link graphs, two-hop friend lists, social network influence analysis [40], and recommender systems [2]. In order to support different graph applications, an efficient large graph processing engine is a must.

Previous studies on building such an engine is to adopt existing distributed data-intensive computing techniques in the cloud [10, 17]. Most of these studies [43, 20, 21] are built on top of MapReduce [10], which is suitable for processing flat data structure, not particularly for graph structured data. More recently, graph systems such as Pregel [29] and Trinity [35] have been developed specifically for large graph processing. Those systems support a vertex-oriented execution model and allow users to develop custom logics on vertices. Take Pregel as an example. Pregel executes user-defined function $Compute()$ per vertex in parallel, based on the general bulk synchronous parallel (BSP) model. By default, the vertices can be stored in different machines according to the simple hash function. However, the simple partitioning function leads to huge network traffic in graph processing tasks. For example, if we want to compute the two-hop friend list for each account in a social network, every friend (vertex) must first send its friends to each of its neighbors, then each vertex combines the friend lists of its neighbors. Implemented with the simple partitioning scheme, this operation results in huge network traffic because of shuffling the vertices.

A traditional way of reducing data shuffling in distributed graph processing is graph partitioning [30, 26, 11]. Graph partitioning minimizes the total number of cross-partition edges among partitions in order to minimize data transfer. The commonly used distributed graph processing algorithms are multi-level algorithms [24, 22, 36]. Those algorithms recursively divide the

graph into multiple partitions with bisections according to different heuristics.

Even the baseline graph processing engine should store the graph into partitions, as opposed to a flat storage. However, there is little attention given to how graph partitioning can be effectively integrated into large processing in the cloud environment. There are a number of challenging issues in the integration. First, graph partitioning itself is a very costly task, generating lots of network traffic. Moreover, partitioned graph storage and vertex-oriented graph processing need careful revisit in the context of cloud. The cloud network environment is significantly different from those in previous studies [24, 22, 26], 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. Current cloud infrastructures are often based on tree topology [14, 5, 19]. 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. Even worse, the topology information is usually not available to users due to virtualization techniques in the cloud. In practice, such network bandwidth unevenness has been confirmed by both cloud providers and users [5, 19]. It requires careful network optimizations and tuning on graph partitioning and processing.

In this paper, we propose a network performance aware graph partitioning framework to improve the network performance of large graph processing on partitioned graphs. The framework improves the network performance of graph partitioning process itself. More importantly, the graph partitions generated from the framework improve the network performance of graph processing tasks. To capture the network bandwidth unevenness, we model the machines chosen for graph processing as a complete undirected graph (namely *machine graph*): each machine as a vertex, and the bandwidth between any two machines as the weight of an edge. The network performance aware framework recursively partitions the data graph, as well as the machine graph, with bisection correspondingly. That is, the bisection on the data graph is performed with the corresponding set of machines selected from the bisection on the machine graph. The recursion terminates when the data graph partition can fit into main memory. By partitioning the data graph and machine graph simultaneously, the number of cross-partition edges among data graph partitions is gracefully adapted to the aggregated amount of bandwidth among machine graph partitions. To exploit the data locality of graph partitions, we develop *hierarchical combination* to exploit network bandwidth unevenness in order to improve the network performance.

We develop a system prototype (named Surfer) following Pregel (the latest vertex-oriented graph engine by Google), and extend it with our graph partitioning framework. We have evaluated the efficiency of Surfer on a real-world social network and synthetic graphs of over 100GB each in a 32-node cluster as well as on Amazon EC2. The experimental results in the local cluster demonstrate that 1) our bandwidth aware graph partitioning scheme improves the partitioning performance by 39–55%, and improves the graph processing by 6–71% under different simulated network topologies; 2) our optimizations reduce the network traffic by 30–95%, and the total execution time by 30–85%. The experimental results on Amazon EC2 shows that our optimizations in Surfer reduce the total execution time by 49% on average.

The rest of the paper is organized as follows. We review the related works on cloud computing and graph processing in Section 2. We present our network performance aware graph partitioning framework in Section 3. We present the experimental results in Section 4, and conclude this paper in Section 5.

2. PRELIMINARY AND RELATED WORK

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

2.1 Cloud computing

A cloud consists of tens of thousands of connected commodity computers. A cloud system often runs in a subset of machines in the cloud. Due to the significant scale, the network environment in the cloud differs with the small-scale cluster. The current cloud practice is to use the switch-based tree structure to interconnect the servers [14]. The key problem of the tree topology is the network bandwidth of any machine pair is not necessarily uniform, depending on the switch connecting the two machines [17]. Moreover, as commodity computers evolve, the cloud evolves and becomes heterogenous among generations [42]. For example, current main-stream network adaptors provide 1Gb/sec, and the future ones with 10Gb/sec. These hardware factors result in the unevenness in the network bandwidth among machines in the cloud.

In addition to hardware factors, software techniques can also result in network bandwidth unevenness. For example, virtual machine consolidation is an effective optimization for the resource utilization of virtualization. Consolidation induces concurrent tasks to compete for the network bandwidth on the same physical machine. Different degrees of consolidation cause the bandwidth unevenness among physical machines.

The unique network environment in the cloud motivates advanced optimizations with the knowledge of network topology (such as multi-level data reduction along the tree topology [10] and partition-based locality optimizations [33]) and scheduling techniques [18]. However, *the topology information in the cloud is usually not available to cloud users due to the virtualization and system management issues.* First, virtualization hides the network topology from users, without exposing the real configurations of the underlying hardware. Second, cloud environments do not offer administrator privileges on the hardware and software under the virtualization layer. Such privileges are usually required for getting the network topology information. This paper develops network-centric optimizations for partitioning and processing a large graph, without the requirement on the knowledge of network topology.

Designing an efficient and user-friendly development platform for applications in the cloud is a hot research topic. A number of cloud systems such as MapReduce [10] (its open-source variant, Hadoop [15]) and Dryad [17] have been developed. The data is stored in the distributed and replicated file system such as GFS [12] or BigTable [7]. 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. More recently, a number of cloud-based data management systems have been developed for data warehousing workloads [39, 1, 16] and on-line transaction processing [9]. All these studies mainly focus on relational data, instead of graph structured data.

2.2 Graph processing

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 (or a partition) of G and V_i to denote the set of vertices in G_i. We define a non-overlapping

partitioning of graph G, to be $\{G_1, G_2, ..., G_k\}$, where $\forall\ i,\ j \in [1...k]$, $k \leq |V|$, $\cup_{i=1}^{k} V_i = V$, $\cup_{i=1}^{k} E_i = E$, $V_i \cap V_j = \emptyset$, 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*.

Large graph processing. Batched processing on large graphs has become hot recently, mainly to meet the requirement on mining and processing those large graphs. Examples include triangle counting [40] and PageRank [31]. Surfer is designed to handle the batched graph processing applications, not transactional graph processing in graph databases like Neo4j and InfiniteGraph etc.

There is some related works on specific tasks on large graph processing in data centers [20, 43, 41, 34]. MapReduce [10] and other systems such as Dryad [17] were applied to evaluate the PageRank for ranking the web graph. HADI [20] and PEGASUS [21] are two recent graph processing implementations based on Hadoop. HADI [20] estimates the diameter of the large graph. PEGASUS [21] supports graph mining operations with a generalization of matrix-vector multiplication. DisG [43] is an ongoing project for the web graph reconstruction using Hadoop. Pujol et al. [34] studied different replication methods to scale the social network analysis. Low et al. [28, 27] presents a new framework with asynchronous programming, which is different from the BSP model in Pregel. Pregel [29] is a BSP-based graph processing engine, with the user-defined API $Compute()$ executed on vertices. In one iteration of BSP (i.e., *superstep* in Pregel's terminology), Pregel executes $Compute()$ on all the vertices in parallel. Hama [4] and Giraph [3] are two open-source projects targeting at large graph processing. They adopt Pregel's programming model but their storage is built on top of HDFS. Trinity [35] is a recent research project in Microsoft, which supports both transactional and batched graph processing. An initial version of Surfer was demonstrated in the comparison with MapReduce [8]. All those systems support the vertex-oriented computation.

Network traffic is a common bottleneck for vertex-oriented computation in those graph processing engines. To the best of our knowledge, this study is the first of its kind to alleviate the network bottleneck of vertex-oriented computation on partitioned graphs in the cloud.

Graph partitioning. Graph partitioning is a well-studied problem in combinatorial optimization with 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 [25]. Karypis et al. [24, 22] proposed a parallel multi-level graph partitioning algorithm, with a minimum bisection on each level. Since bisection is commonly used in multi-level graph partitioning algorithms, this paper considers bisections. However, the graph partitioning framework can be easily extended to k-section based graph partitioning algorithms.

Since graph bisection has been a key operation in multi-level graph partitioning [24, 22], we briefly introduce the process of bisection. There are three phases in a graph bisection, namely *coarsening*, *partitioning* and *uncoarsening*, as illustrated in Figure 1. The coarsening phase consists of multiple iterations. In each iteration, multiple adjacent 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) [23]. In the uncoarsening phase, the partitions are then iteratively projected back towards the original graph, with a local refinement on each iteration. The iterations are highly parallelizable, and their efficiency and scalability has been evaluated on shared-memory architectures (such as Cray supercomputers) [24, 22]. However, in the coarsening and uncoarsening phases, all the edges may be accessed, generating a lot of network traffic if the input graph is stored in distributed machines.

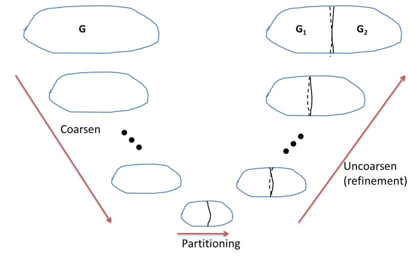


Figure 1: The three phases in graph bisection: *coarsening*, *partitioning* and *uncoarsening*.

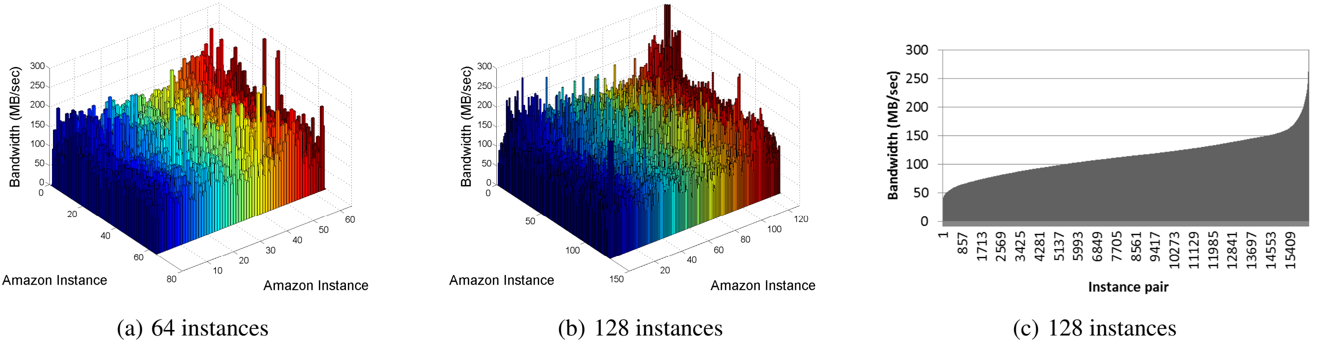
Existing distributed and parallel graph partitioning algorithms such as ParMetis [30] are suboptimal in the cloud. In particular, they do not consider the unevenness of the network bandwidth in the cloud. While they have demonstrated very good performance on shared-memory architectures [24], unevenness of network bandwidth results in deficiency in partitioning itself and also the efficiency of partitioned graph storage and processing. 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 develop a network performance aware framework to adapt multi-level graph partitioning [24, 22] to the network environment in the cloud.

3. GRAPH PARTITIONING FRAMEWORK

In this section, we start with the motivations for network performance aware optimizations. Next, we present the models for multi-level graph partitioning and for the network performance in the cloud environment. Finally, we present our network performance aware framework for graph partitioning. In the next section, we present the evaluation results on Surfer, which extends Pregel with the proposed graph partitioning framework.

3.1 Motivations

Surfer is a master-slave system, consisting of one master server and many slave servers. The slave servers store graph partitions and perform graph computation. We study the factors affecting the network performance of graph processing. We assume that the amount of network traffic sent along each cross-partition edge is the same (denoted as b). Denote the number of cross-partition edges from partition G_i to G_j to be $C(G_i, G_j)$, and the network bandwidth between the machines stored G_i and G_j to be $B_{i,j}$. Since network bandwidth is a scarce resource in the cloud environment [10, 17], we consider the bandwidth as the main indicator for network performance, and approximate the network data transfer time from G_i to G_j to be $\frac{C(G_i,G_j)\times b}{B_{i,j}}$. This approximation is sufficient for large graph processing in



(a) 64 instances (b) 128 instances (c) 128 instances

Figure 2: Network bandwidth unevenness in Amazon EC2: (a, b) Pair-wise network bandwidth varying the number of small instances, (c) the distribution of pair-wise network bandwidth. The y-axis of all figures is capped at 300 for clarity.

both private and public cloud environments, as we observed in our experiments. Assuming P graph partitions are stored on P different machines, the total network data transfer time incurred in all partition pairs is $\sum_{i=0}^{P-1}\sum_{j=0}^{P-1}\frac{C(G_i,G_j)\times b}{B_{i,j}}$.

Clearly, if the network bandwidth among different machine pairs ($B_{i,j}$, $\forall i,j < P$) is constant, minimizing the total number of cross-partition edges also minimizes the total network data transfer time. However, the network bandwidth among different machine pairs can vary significantly in the cloud. Such network bandwidth unevenness has been observed by cloud providers [5, 19]. We have also observed significant network bandwidth unevenness in Amazon EC2. Figure 2 shows the network bandwidth of every machine pair among 64 and 128 *small* instances (i.e., virtual machine) on Amazon EC2. The network bandwidth varies significantly. The mean (MB/sec) and standard deviation are (112.8, 37.5) and (115.0, 40.2) for 64 and 128 small instances, respectively. We observed that some pair-wise bandwidth is very high (e.g., more than 500 MB/sec). The possible reason is that those small instances can be allocated to the same physical machine.

We also note that the network bandwidth between two instances in the public cloud is temporally stable, with similar results observed in the previous study [37]. That allows us to maintain the network bandwidth for a machine pair with a reasonable long period, and to develop network performance aware optimizations for graph processing.

Due to the network bandwidth unevenness, the way of partitioning and storing graph partitions on the machines is an important factor for the efficiency of graph processing. Since the number of graph partitions and the number of machines for graph processing can be very large, the possible solution space of storing graph partitions to the machines is huge. Consider P partitions to be stored on P machines. The space includes P! possible solutions. Another problem is how to make the graph partitioning and graph processing algorithm aware of the bandwidth unevenness for networking efficiency.

To address the network bandwidth unevenness in the current cloud environment, we propose a network performance aware framework for graph partitioning and graph processing. The basic idea is to partition, store and process 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. This is because the network traffic requirement for those graph partitions is high.

3.2 Models

We use two models namely *partition sketch* and *machine graph* to capture the features of graph partitioning process and network performance, respectively.

3.2.1 *Partition Sketch*

We model the process of a multi-level graph partitioning algorithm as a tree structure (namely *partition sketch*). Each node in the partition sketch represents the graph acting as the input for the partition operation at a level of the entire graph partitioning process: the root node representing the input graph; non-leaf nodes at level $(i+1)$ representing the partitions of the ith iteration; the leaf nodes representing the graph partitions generated by the multi-level graph partitioning algorithm. The partition sketch is a k-ary tree for k-section based graph partitioning algorithm. Since this paper mainly considers bisections, the partition sketch is represented as a binary tree. If the number of graph partitions is P, the number of levels of the partition sketch is $(\lceil\log_2 P\rceil + 1)$.

Figure 3 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 [30, 24, 22] can achieve. Partitioning with optimal bisections does not necessarily result in P partitions with the globally minimum number of cross-partition edges. Existing studies [24, 22] have demonstrated that they can achieve relatively good partitioning quality, approaching the global optimum. Thus, we use the ideal partition sketch to study the properties of the multi-level partitioning algorithm.

Analyzing the graph partitioning process, we have found that 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.

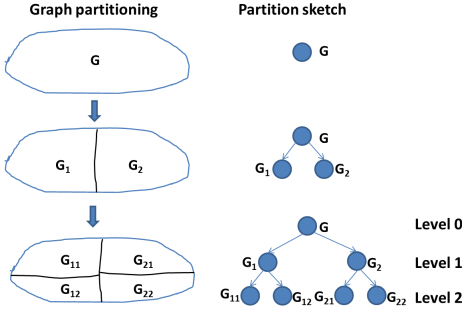


Figure 3: Correspondence between bisections and the partition sketch for the process of partitioning the graph into four partitions.

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 increase in the number of cross-partition edges in the recursive partitioning process.

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).

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:

$$C(n_1, n_2) + C(n_1, n_4) + C(n_3, n_2) + C(n_3, n_4) \geq C(p, p') \quad (1)$$

$$C(n_1, n_2) + C(n_1, n_3) + C(n_4, n_2) + C(n_4, n_3) \geq C(p, p') \quad (2)$$

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

$$C(n_1, n_3) + C(n_2, n_4) \leq C(n_1, n_2) + C(n_3, n_4) \quad (3)$$

$$C(n_2, n_3) + C(n_1, n_4) \leq C(n_1, n_2) + C(n_3, n_4) \quad (4)$$

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).

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 graph partitioning and processing, in order to match the network bandwidth with the number of cross-partition edges.

P_1. Graph partitioning and processing 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 total bandwidth between the two machine sets is the lowest.

P_2. The partition size should be carefully chosen for the efficiency of processing. The number of partitions should be no smaller than the number of machines available for parallelism. 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 main memory of a machine, 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.

3.2.2 *Machine Graph Building*

Graph partitioning and processing are usually performed on a set of machines (or virtual machines) acquired from the cloud provider. We model the machines used for processing the data graph as a weighted graph (namely *machine graph*). In a machine graph, each vertex represents a machine. We assume that each machine has the same configuration in terms of computation power and main memory. In practice, users usually acquire the virtual machines of the same type for one application, because of convenience and management. We consider handling heterogenous machines as future work. 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 bandwidth is similar in both directions. We use the following techniques to build the machine graph without knowledge of network physical topology.

Given a set of machines for partitioning, the machine graph can be easily constructed by calibrating the network bandwidth between any two machines in the set. In our experiments, we measure the network bandwidth by sending a data chunk of 8MB. We use the average of twenty measurements. For N virtual machines, we need N iterations of calibrations in order to get all pair-wise performance. In each iteration, $\frac{N}{2}$ machine pairs are calibrated. The maintenance is based on the classic exponential average, by getting the bandwidth of data transfer in the graph processing.

The left part of Figure 4(a) illustrates the machine graph for four machines in a cluster with tree topology. The edge thickness represents the weight: a thicker edge means a link with higher bandwidth. 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 intra-pod connections are represented as thicker edges, indicating that they have a higher interconnected bandwidth.

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 data graph for partitioning, respectively.

3.3 Bandwidth Aware Graph Partitioning

With the partition sketch and the machine graph in hand, we develop a novel network bandwidth aware framework for graph partitioning and processing in the cloud. The framework enhances a common multi-level graph partitioning algorithm with the network performance awareness. Given a set of machines to partition the graph, the graph is initially stored in those machines (usually according to the simple hash function). At each bisection, all edges and vertices are accessed multiple times for coarsening and uncoarsening. It generates a lot of network traffic. Thus, bisection should be designed to be aware of the network bandwidth unevenness.

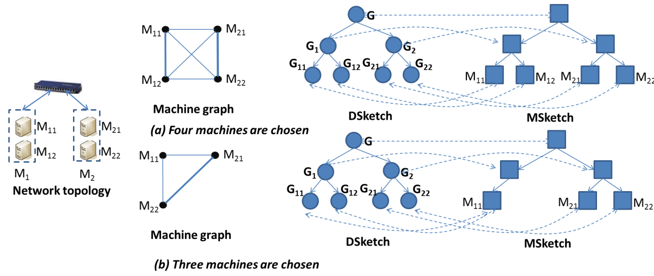


Figure 4: Mapping on the partition sketches between the machine graph and the data graph.

There is one observation on multi-level graph partitioning algorithms: *due to the divide-and-conquer nature, there is no data exchange between the two bisection sub-partitions generated from the same bisection.* Suppose a distinct subset of machines is responsible for each of the two sub-partitions. The network connections between the two subsets of machines are no longer involved in the deeper level of the bisection. That means, we should pick the high bandwidth connections remaining in the subset of machines, and leave the low bandwidth connections as those between the two subsets of machines. This is analogous to performing graph partitioning on the machine graph with respect to minimizing the total bandwidth between two subsets of machines. That results in the correspondence between partitioning the data graph and partitioning the machine graph, and we gradually assign the subset of machines that are suitable to handle graph partitioning at a certain level. At each level of graph partitioning, the framework partitions the data graph and machine graph simultaneously, and matches the network bandwidth in the cloud to the number of cross-partition edges according to the partition sketch and the machine graph.

The bandwidth aware graph partitioning framework is shown in Algorithm 1. The framework simultaneously partitions the data graph and the machine graph with multi-level bisections. At a certain level, it assigns the machines in a partition of the machine graph to perform partitioning on the partition of the data graph. At the leaf level, graph partitions are stored in the machine in the corresponding node in the machine graph. Finally, the partition sketches for both machine graph and data graph are generated. In Surfer, those partition sketches are stored as catalog data in the master server.

The number of partitions, P, can be specified by the user. In Surfer, we determine P so that a graph partition can fit into the main memory of a machine. This is to avoid the significant performance degradation due to the random disk I/O in graph processing. A machine can hold more than one graph partition. In Line 4 of Procedure *BAPart(M, G, l)*, M consists of a single machine. We further divide G into 2^{L-l} partitions so that each partition can fit into the main memory.

We use a local graph partitioning algorithm such as Metis [30] to partition the machine graph, since 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 selected machines. 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

Algorithm 1 Bandwidth aware graph partitioning

Input: A set of machines S in the cloud, the data graph G, the number of partitions P ($L = \log_2 P$)
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: Divide M into two partitions M_1 and M_2;
8: Divide G into two partitions G_1 and G_2 with the machines in M with distributed algorithm [22];
9: *BAPart*(M_1, G_1, l+1);
10: *BAPart*(M_2, G_2, l+1);

adaptation on assigning the network bandwidth to partitions with different number of cross-partition edges. The constraint of making partitions with the roughly 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 machines and the partitions. The mapping guides the machines where the graph partition is further partitioned, and where the graph partition is stored. Figure 4 illustrates the mapping between two machine graphs and a data graph for the partitioning framework. Take case (a) where four machines are selected as an example. The bisection on the entire graph G is done on all the four machines. 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. Note, we use the tree structured network topology mainly for presentation purposes, and our framework does not assume specific network topologies and network environments.

The partitioning algorithm satisfies the three design principles: 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.

3.4 Partitioned Graph Storage

We consider how graph partitions are distributed on the machines so that the network performance is optimized. The distribution is also maintained without re-partitioning when the machine graph is significantly changed. In Surfer, graph partitions are stored securely with the replication scheme. In Amazon EC2, we store graph partitions in Amazon S3 which offers secure storage by default. For processing, we read the graph partitions from Amazon S3, and store them on the local storage of virtual machines that we have acquired for graph processing. If such a secure storage is not available, one can develop a distributed file system like GFS [12]. In both cases, graph partitions are stored in the local storage of machines for graph processing.

We transform the problem of distributing graph partitions to the machines as the problem of developing a node-to-node mapping from the partition sketch of the data graph ($DSketch$) to the partition sketch of the machine graph ($MSketch$). If a node n in $DSketch$ is mapped to a node n' in $MSketch$, all the partitions generated from n is stored in the machines in n'. If n' represents a single machine, the mapping gives all the graph partitions stored in that machine. $DSketch$ and $MSketch$ can be obtained from catalog.

The node-to-node mapping from $DSketch$ to $MSketch$ is defined as follows. Let the number of levels in $DSketch$ and $MSketch$ be L_D and L_M, respectively. We assume $L_D \geq L_M$ so that each machine has one graph partition at least. Let R_D and R_M be the root nodes of $DSketch$ and $MSketch$, respectively. We define a mapping $\mathcal{M}$ from $DSketch$ to $MSketch$ in a divide-and-conquer manner (the below equation), where d_1 and d_2 is the left and the right subtrees of R_D respectively, and m_1 and m_2 is the left and the right subtrees of R_M respectively. d_1 (or d_2) can be mapped to any of m_1 and m_2. Thus, we use "|" to indicate that both $\mathcal{M}$ mappings are valid.

$$\mathcal{M}(R_D, R_M) = \begin{cases} \mathcal{M}(d_1, m_1), \mathcal{M}(d_2, m_2)| & \\ \mathcal{M}(d_1, m_2), \mathcal{M}(d_2, m_1) & \text{if } R_D and R_M are not leaves, \\ R_D \rightarrow R_M & otherwise \end{cases}$$

The deepest level of $\mathcal{M}$ ($d \rightarrow m$) means the partitions generated from further partitioning d are stored on machine m. Figures 4 (a,b) illustrate the mappings from $DSketch$ and $MSketch$ when the number of machines are four and three, respectively. When the number of machines is four, each graph partition is mapped to different machines. When the number of machines is three, two graph partitions in G_1 are mapped to a single machine.

According to the definition, there is $\prod_{i=1}^{L_M} 2^{i-1}$ possible mapping candidates. In initialization, we can pick any one of them. In maintenance, we choose the one that results in the minimum network traffic. Also note, the mapping obtained from Algorithm 1 satisfies this definition. Thus, if users acquire a different set of machines, we only need to construct $MSketch$ and calculate the mapping (with the $DSketch$ from the catalog). That is, we can easily run graph processing on different sets of machines, without repartitioning the data graph.

The mapping from $DSketch$ to $MSketch$ needs to be maintained. In practice, data graphs are not often re-partitioned (i.e., $DSketch$es are usually static). We need to maintain $MSketch$ if the machine graph is significantly changed, for example, a machine failure occurs. In particular, we periodically check the machine graph to see whether we need to perform adjustment on the graph partitions. For simplicity, we decide to perform adjustment if the cross-machine bandwidth significantly changes (e.g., the total amount of bandwidth changes in the bandwidth is higher than a predefined threshold), or when the set of machines changes. In case where $MSketch$ is reconstructed, we consider all the possible mappings from $DSketch$ to $MSketch$, and choose the one with the smallest number of graph partition movements. Specifically, given a mapping, we can obtain the graph partitions on each machine and calculate the number of graph partition movements by comparing the current graph partition distribution. Thus, we are able to shuffle the affected graph partitions only, without entirely repartitioning the data graph.

The proximity of the data graph partitions is gracefully adapted to the the network bandwidth of the machines during the mapping process. That is, the total number of cross-partition edges in the bisection represented by $DSketch$ corresponds to the total bandwidth of the connections in the cut of the bisection represented by $MSketch$. Moreover, the partitions stored in a machine have the same lowest common ancestor node in $DSketch$.

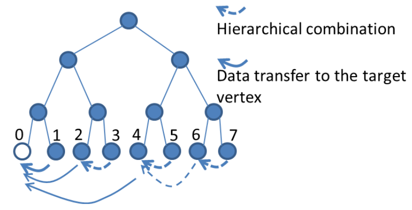


Figure 5: Hierarchical combination according to the partition sketch of the machine graph of eight machines.

3.5 Hierarchical Combination

We extend the vertex-oriented execution model to exploit data locality of graph partitions. Since combination is a commonly used approach to reduce network traffic in data intensive computing systems [10, 17], we develop the combination optimization on partitioned graph. The basic idea is to apply a $Combine()$ function (i.e., *Combiner* in Pregel), and perform partial merging of the intermediate data before they are sent over the network. Combination is applicable when the combination function is annotated as an associative and commutative function.

A basic approach is *local combination*. Current graph engines like Pregel and Trinity support this basic approach. For all the graph partitions on a machine, we apply 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 further processing.

Local combination is not aware of the network bandwidth unevenness in the cloud network environment. This motivates us to develop the network performance aware optimization for graph processing, i.e., *hierarchical combination*. In local combination, it requires network data transfer for the boundary vertices of the graph partition. Due to the irregular graph structures, the source vertices are likely to be scattered on many different machines. Thus, many data transfers are performed on the relatively low bandwidth machine pairs, caused by the network bandwidth unevenness.

Instead of direct data transfers after local combination, the data of the source vertices can be combined among the machines with high bandwidth before sending them to the target machine via the connections with low bandwidth. Hierarchical combination applies this idea in multiple levels according to the partition sketch of the machine graph. With the hierarchical combination optimization, the data transfer on the low-bandwidth connection is reduced. We note that similar optimization techniques have been adopted in other contexts to reduce network traffic, e.g., MPI collective communications [32]. Differently, the proposed hierarchical combination is guided by the partition sketches of the machine graph and the data graph, which are specifically designed for graph processing on partitioned graphs.

Figure 5 illustrates one example of performing hierarchical combination on eight machines. Suppose each machine holds one graph partition and machine 0 needs to read data from other machines. Note that, the partition sketch of the machine graph has captured the network bandwidth unevenness. After local combination on each machine, we perform the first-level combination between two machines (for example, between machines 6 and 7), and store the result on a *representative* machine. Machines 2, 4 and 6 are the representative machines at the first-level combination. Further

combination is performed on the representative machines. Finally, all the partial results are sent to machine 0. On the low-bandwidth connections between machine 0 and machine i ($4 \leq i \leq 7$), hierarchical combination has only one data transfer for the partial results, compared with four in the baseline implementation with local combination.

4. EVALUATION

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

4.1 Experimental Setup

We have conducted our experiments on a local cluster and Amazon EC2. The local cluster has 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. All the machines form a pod, sharing the same switch. The current cluster provides even network bandwidth between any two machines.

We develop a system prototype (named Surfer) following Pregel [29], and extend it with our graph partitioning framework. We implement Surfer in C++, compiled in Visual Studio 9 with full optimizations enabled.

We have described some implementation details on graph partitions. The data graph is divided into many partitions with similar sizes, using our graph partitioning framework (Section 3). Graph partitions are stored in the hard disk (such as local storage in Amazon EC2), and a partition is loaded into main memory when the task is initiated for processing the partition. The local cluster adopts a distributed file system with replications (each graph partition has three replicas by default).

Surfer uses the adjacency list storage as graph storage. Other graph storage formats are also applicable. 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. Instead of maintaining a global mapping from an arbitrary vertex ID to its partition ID, we encode the vertex IDs such that the partition ID could be inferred from vertex ID itself. The vertex ID is divided into two bits ranges, the higher range represents its partition ID, and the lower stands for its offset id within this partition. From this encoding, it is straightforward to find the partition ID for a vertex.

Simulating different network environments. While Amazon EC2 allows us to evaluate Surfer in a public cloud environment, the network topology is hidden by virtualization. We need another complementary approach to evaluate our framework in a controlled manner. We simulate the network bandwidth unevenness in the cloud network environment. 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 data transfer time obtained from network simulator. We denote the setting of the current cluster to be T_1. We consider the following two settings T_2 and T_3. The 32 machines are simulated as a subset of machines in a much larger cloud.

Since tree-structured network is the major topology in current data centers [14, 5, 19], we simulate different tree-structured network topologies. We simulate T_2 as a tree topology. 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. At each level of the tree topology, all-to-all communications in graph processing cause the traffic contention in the switch [14].

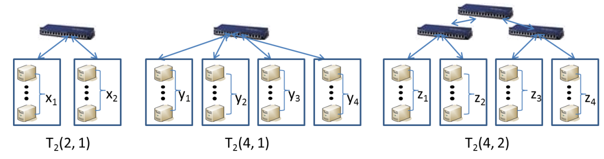


Figure 6: The variants of topology T_2, simulating the machines in the cloud with different tree network topologies. $\sum_{i=1}^{2} x_i = \sum_{i=1}^{4} y_i = \sum_{i=1}^{4} z_i = 32$, $x_i \geq 1$ ($1 \leq i \leq 2$), $y_j \geq 1$, $z_j \geq 1$ ($1 \leq j \leq 4$). By default, $T_2(2,1)$: $x_1 = x_2 = 16$; $T_2(4,1)$: $y_1 = y_2 ... = y_4 = 8$; $T_2(4,2)$: $z_1 = z_2 ... = z_4 = 8$.

Figure 6 shows the three variants of T_2 with 32 machines examined in our experiments. We can configure the number of machines in different pods ($T_2(2,1)$: $x_i \geq 1$ ($1 \leq i \leq 2$), $T_2(4,1)$: $y_j \geq 1$, $T_2(4,2)$: $z_j \geq 1$ ($1 \leq j \leq 4$)). By default, we assign the same number of machines in each pod, and all the switches have the bandwidth of 1Gb/sec. Our experiments are conducted without the knowledge of network topologies. For different numbers of pods and different distributions of machines among pods, we observed that the machine graph model always correctly captures the network bandwidth unevenness in the 32 machines.

T_3 simulate a pod whose machines have two different configurations. For simplicity, we simulate q ($1 \leq q < 32$) machines randomly chosen from the pod having one half bandwidth of the remainder machines in the cluster.

We develop a discrete-event network simulator for T_2 and T_3. We log the events of data send/receive operations in the execution of Surfer, and then feed those events into the simulator to get the delay of each send/receive operations. Next, we use those latency information in the real execution by adding a latency into the send/receive operations such that the time for sending the data matches the simulation on the target tree topology.

Graph operations. We consider multiple common operations in social network which represent basic processing on graphs [40, 2]. We can find their counterparts in other graph applications such as web graph analysis.

*Network ranking (*NR*)* is to generate a ranking on the vertices in the graph using PageRank [31] or its variants.

*Recommender system (*RS*)* is to evaluate 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.

*Triangle counting (*TC*).* Previous studies [40] 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 social network. Triangle counting requires a single iteration for propagation on the graph.

*Vertex Degree Distribution (*VDD*)* calculates the out-degree distribution of a social graph.

*Reverse Link Graph (*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.

*Two-hop Friends List (*TFL*)* finds the list of two-hop friends for analyzing social influence spread, community detection and so on. The ratio of the selected vertices is 10% in our experiments.

These graph operations have different characteristics to assess different system aspects. VDD is a vertex-oriented tasks, and others

Table 1: **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

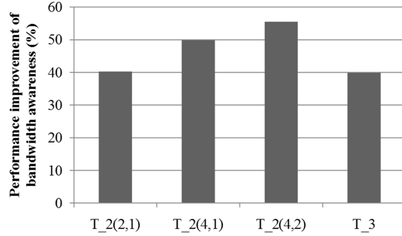


Figure 7: Performance improvement of bandwidth awareness on graph partitioning over ParMetis.

are edge-oriented tasks. TFL has a heavy data transfer among neighbor vertices, and NR has a relatively light data transfer.

Data sets. The data sets include a snapshot of the social network in MSN collected in 2007 and synthetic graphs, each of which is over 100GB. The social network used in this study contains 508.7 millions vertices and 29.6 billion edges. The number of edges in the social network is almost five times as many as the largest one in the previous study [21].

We generate synthetic graphs simulating small world phenomenon. We first generate multiple small graphs with small-world characteristics using an existing generator [6], 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 to evaluate the scalability of Surfer. The default size is 100GB, with 408.4 million vertices and 25.9 billion edges.

Metrics. 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 job till its completion, and the total machine time is the total time spent on the entire job 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 the execution.

We run the same experiments for five times and report the average execution time. The results across different runs are mostly stable in our simulation. We obtain similar results for the real graph and synthetic graphs. We mainly present our evaluation results on the real-world social network, and the results for parametric studies on the synthetic graphs. We report the results for single iteration only, unless specified otherwise. Also, the reported results are mainly from the local cluster, and the results for Amazon EC2 are presented in Section 4.6.

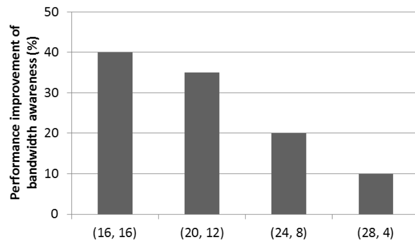


Figure 8: Performance improvement of bandwidth awareness on graph partitioning over ParMetis on varying $T_2(2,1)$.

4.2 Results on Partitioning

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 1 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 depth 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 graph partition and the intermediate data usually cannot fit into main memory, and cause a huge amount of random disk I/Os. Therefore, we choose 2GB as our default setting, and divide the real graph into 64 partitions. We use this setting for the real graph thereafter.

Performance improvement on graph partitioning. We next evaluate the effectiveness of our bandwidth aware graph partitioning framework. We investigate the improvement on the elapsed time of a well-known distributed graph partitioning software namely ParMetis [30] on T_2 and T_3 when our framework is used, as shown in Figure 7. Our framework has the same performance as ParMetis on T_1.

On different network environments (T_2 and T_3), the bandwidth aware graph partitioning framework achieves an improvement of 39–55% over ParMetis. ParMetis randomly chooses the available machine for processing, without the awareness of the network bandwidth unevenness. In contrast, due to the network performance aware optimization, our framework effectively utilizes the network bandwidth, and reduces the elapsed time of partitioning. The performance improvement increases for the topology T_2 with more levels or with more pods per level. This demonstrates the effectiveness of the three design principles of an efficient graph partitioning algorithm. Note, both techniques on T_1 behave the same, since every machine pair in T_1 has almost the same network bandwidth.

We further study the impact of different distributions of machines among pods. Figure 8 shows the performance improvement of graph partitioning varying (x_1, x_2) on $T_2(2,1)$. As x_1 increases from 16 to 28, one pod has more machines and the network bandwidth becomes more even among different machine pairs. The performance improvement degrades. When $x_1 = 28$, the performance improvement is around 10%. We observed similar results on other network topologies.

4.3 Results on Graph Processing

We first study the impact of optimization techniques in the execution in a single-iteration Surfer. We have studied the case for multiple iterations, and obtained similar results. Tables 2 and 3 show the timing and I/O metrics on the applications on T_1. On T_1, hierarchical combination degrades to the local combination within individual graph partitions. We evaluate the impact of hierarchical combination in the later experiment. We implement these applications with Surfer in the following optimization levels.

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

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**

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

O3. Surfer with local combination, but with graph partition storage distribution generated from ParMetis.

O4. Surfer with local combination, 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' graph partition distribution and our bandwidth aware framework. The difference between O1 and O3 as well as between O2 and O4 is to evaluate the effectiveness of local combination 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, the performance improvement is 3–17%. When both techniques are enabled, the performance improvement is better, between 6% and 29%. This measures the performance improvement of Surfer when both optimization techniques enabled. 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 same machine. Surfer schedules the execution according to the partition sketch and takes advantage of such locality. Since VDD is a vertex-oriented task, bandwidth aware graph partitioning has little improvement.

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

Therefore, comparing O1 with O4, the storage distribution and local combination are accumulative. Their combined performance improvement is 30–85% and the combined network traffic reduction is 30–95% (except VDD), which also represents the performance improvement of Surfer over the basic performance with the partitioned graph. 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 combination significantly reduces the data transfer, especially when the data distribution is according to the bandwidth aware framework.

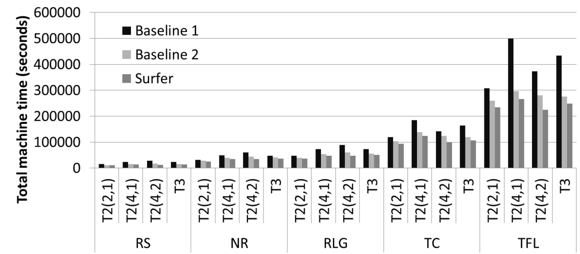


Figure 9: Impact of bandwidth aware partitioning on different topologies

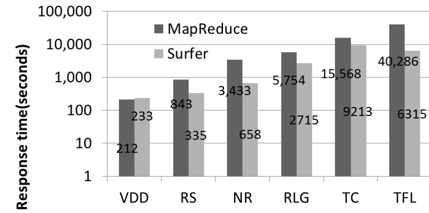


Figure 10: Performance comparison between MapReduce and Surfer

Impact of hierarchical combination. We further investigate the impact of hierarchical combination on different network environments. We denote "Baseline 1" to be the baseline Surfer with O3, and "Baseline 2" to be "Baseline 1" with the bandwidth aware graph partitioning, without hierarchical combination. Thus, the performance difference between Baseline 1 and Baseline 2 is the impact of bandwidth aware graph partitioning, and the performance difference between Baseline 2 and Surfer (with full optimizations) measures the impact of hierarchical combination when bandwidth aware graph partitioning is enabled.

Figure 9 shows the performance of Surfer in comparison with Baseline 1 and Baseline 2 on T_2 and T_3. Bandwidth aware graph partitioning significantly improves the performance on different network topologies, with an improvement up to 60%. With the awareness of bandwidth unevenness, the hierarchical combination further improves Baseline 2 by 10-12%.

We further study the impact of different distributions of machines among pods in those network topologies (Figures are omit-

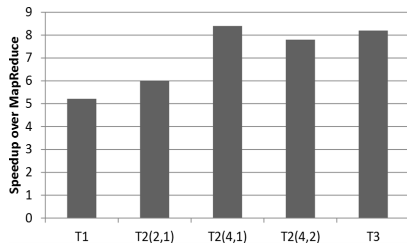


Figure 11: Comparison of network ranking between MapReduce and Surfer

ted). Generally, as one pod has more machines, the performance impact of our graph partitioning framework degrades. Without the knowledge of network topologies, the proposed graph partitioning framework captures the network bandwidth unevenness for improving the overall performance of graph processing. This is consistent with the results on graph partitioning.

4.4 Comparisons with MapReduce

For the completeness of our study, we have compared Surfer with MapReduce-based implementations. Figure 10 shows the performance comparison between MapReduce and Surfer on the applications. The MapReduce engine is built on top of Dryad [17]. Surfer 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, which confirms the importance of developing graph-aware engines for large graph processing [29, 21].

We further study the performance comparison between MapReduce and Surfer for network ranking under different simulated topologies. Figure 11 shows the speedup (defined as $\frac{t_s}{t_m}$, where t_s and t_m are the response time for Surfer and MapReduce, respectively). The MapReduce implementation does not have the network bandwidth aware optimizations. Our network bandwidth aware optimizations in Surfer further increase the performance speedup over MapReduce on T_2 and T_3.

4.5 Other System Features

Machine failure. Figure 12 shows the disk I/O rates of an execution of the network ranking, where we intentionally kill a machine at 235 seconds (as shown in Figure 12(a)). Upon detecting the machine failure, Surfer decides to re-construct the machine graph and to maintain the mapping from the partition sketch of data graph to the partition sketch of machine graph. Two graph partitions are replicated due to the replication protocol. When the maintenance is finished, Surfer immediately schedules the failed task for re-execution. The re-execution happens in another machine (shown in Figure 12(b)). Comparing the normal execution in Figure 12(c), the entire computation with recovery finishes in 723 seconds including a startup overhead of 10% over the normal execution.

4.6 Results on Amazon EC2

We further study the network performance aware graph partitioning and graph processing on Amazon EC2. Figure 13 (a) shows the performance improvement of our network bandwidth aware optimization on graph partitioning, and Figure 13 (b) compares the response time of NR with different approaches. We increase the number of medium instances from 32 to 128 and meanwhile increase the size of synthetic graphs from 25GB to 100GB. We measure 100 times of each experiment on the same set of instances, and report the average and the range for the elapsed time of graph partitioning and processing. The variation is acceptable in Amazon EC2. Due to the network bandwidth unevenness in Amazon EC2, our network performance aware optimizations improve both graph partitioning and processing, with 20–25% performance improvement for graph partitioning and with 49% and 18% performance improvement for NR over Baseline 1 and 2 respectively. This demonstrates the effectiveness of our network performance aware optimizations in the public cloud environment. We also performed the same experiment on other instances on Amazon EC2 and observed similar results (figures are omitted).

5. CONCLUSION

As data-intensive applications on large graphs become popular in the cloud, the efficiency of large graph processing engines needs to be carefully revisited. In this paper, we examine the unique network environment in the cloud, and develop a bandwidth aware graph partitioning framework to minimize the network traffic in partitioning and processing. We develop Surfer by extending a system prototype following Pregel with the graph partitioning framework. Our evaluation on the large real-world and synthetic graphs shows the effectiveness of the bandwidth aware graph partitioning and processing optimizations on Surfer. We expect that our graph partitioning framework is applicable to other vertex-oriented graph processing engines such as Trinity [35]. Our future work is to explore monetary cost optimizations for graph processing [38], and to investigate the performance of Surfer on other network environments [14].

Acknowledgement

The authors would like to thank anonymous reviewers for their valuable comments. The work of Rishan Chen, Xuetian Weng and Xiaoming Li was partly supported by NSFC Grant 60933004 and SKLSDE-2010KF-03 of State Key Laboratory of Software Development Environment at Beihang University. The work of Bingsheng He was partly supported by SUG Grant M4080102.020 of Nanyang Technological University, Singapore.

6. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2009.

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

[3] Apache Giraph. *http://giraph.apache.org/*.

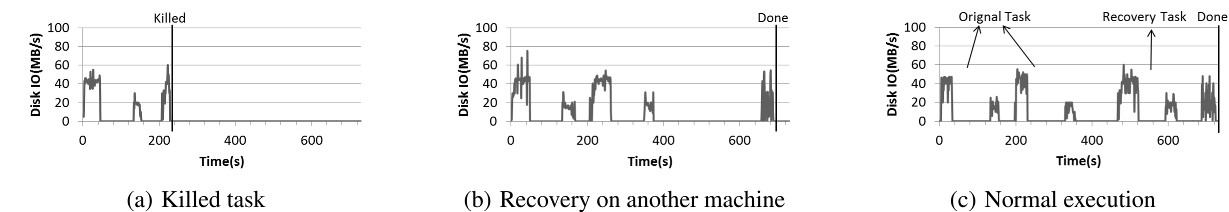
[4] Apache Hama. *http://hama.apache.org/*.

[5] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[6] 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.

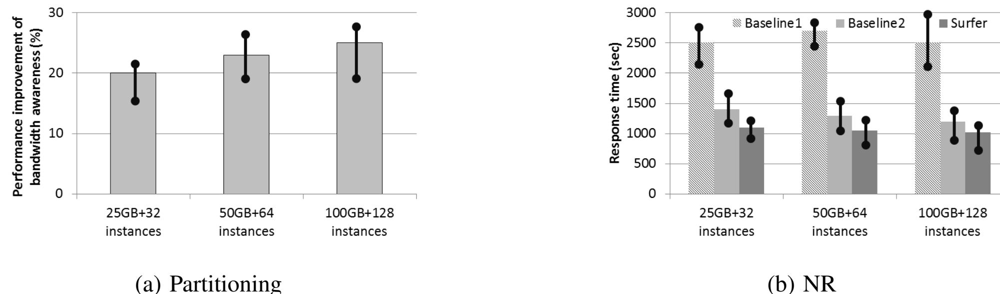
[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006.

[8] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1123–1126, New York, NY, USA, 2010. ACM.



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

Figure 12: Disk I/O rates rates over time for different executions of the NR



(a) Partitioning (b) NR

Figure 13: Network performance aware graph partitioning and NR on Amazon EC2 with the number of medium instances varied

[9] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC: ACM symposium on Cloud computing*, 2010.

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

[11] B. Derbel, M. Mosbah, and A. Zemmari. Fast distributed graph partition and application. In *IPDPS*, 2006.

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

[13] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[14] 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.

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

[16] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 63–74, New York, NY, USA, 2010. ACM.

[17] 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.

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

[19] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC*, 2009.

[20] 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, CMU, 2008.

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

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

[23] 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.

[24] 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.

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

[26] S. Koranne. A distributed algorithm for k-way graph partitioning. In *EUROMICRO*, 1999.

[27] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[28] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, 2010.

[29] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

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

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

[32] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, June 2007.

[33] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[34] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *SIGCOMM*, 2010.

[35] B. Shao, H. Wang, and Y. Li. The trinity graph engine. Technical Report 161291, Microsoft Research, 2012.

[36] A. Trifunović and W. J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel Distrib. Comput.*, 68, May 2008.

[37] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM*, 2010.

[38] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou. Distributed systems meet economics: pricing in the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.

[39] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *SIGMOD*, 2010.

[40] 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.

[41] W. Xue, J. Shi, and B. Yang. X-rime: Hadoop based large scale social network analysis. In *SCC*, 2010.

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

[43] 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.