DD-Graph: A Highly Cost-Effective Distributed Disk-based Graph-Processing Framework

YongLi Cheng[†] Fang Wang[§†] Hong Jiang[‡] Yu Hua[†] Dan Feng[†] XiuNeng Wang[†]
[†]School of Computer, Huazhong University of Science and Technology, Wuhan, China
[†]Wuhan National Lab for Optoelectronics, Wuhan, China
[‡]Department of Computer Science & Engineering, University of Texas at Arlington, USA
{chengyongli,wangfang}@hust.edu.cn,hong.jiang@uta.edu,{csyhua,dfeng,xiunengwang}@hust.edu.cn
[§]Corresponding Author: Fang Wang

ABSTRACT

Existing distributed graph-processing frameworks, e.g., GPS, Pregel and Giraph, handle large-scale graphs in the memory of clusters built of commodity compute nodes for better scalability and performance. While capable of scaling out according to the size of graphs up to thousands of compute nodes, for graphs beyond a certain size, these frameworks usually require the investments of machines that are either beyond the financial capability of or unprofitable for most small and medium-sized organizations. At the other end of the spectrum of graph-processing frameworks research, the single-node disk-based graph-processing frameworks, e.g., GraphChi, handle large-scale graphs on one commodity computer, leading to high efficiency in the use of hardware but at the cost of low user performance and limited scalability. Motivated by this dichotomy, in this paper we propose a distributed disk-based graph-processing framework, called DD-Graph, that can process super-large graphs on a small cluster while achieving the high performance of existing distributed in-memory graph-processing frameworks.

Keywords

Cost-Effectiveness; High Performance; Super-Large Graphs

1. INTRODUCTION

With the rapid growth of data, there has been a recent surge of interest in processing large graphs in both academia and industry. Due to the fact that many graph algorithms exhibit irregular access patterns [2], most graph processing frameworks require that the graphs fit entirely in memory, necessitating either a supercomputer or a very large cluster to process large graphs [3, 5].

Several graph-processing frameworks, e.g., GraphChi [1] and XStream [9], have been proposed to process graphs with billions of edges on just one commodity computer, by relying on secondary storage [1, 9]. However, the performance of these frameworks is limited by the limited secondary storage bandwidth of a single compute node [4] and the significant difference in the access speeds between secondary storage and main-memory [7]. Furthermore, the limited amount of storage of a single commodity computer can potentially limit the scale of the processed graphs, since graphs continue to grow in size [6].

The key difference between the distributed in-memory graph-processing frameworks and single-node secondary storage based graph-processing frameworks lies in the trade-off between the hardware cost and performance, with the former trading off hardware cost for performance while the latter doing the exact opposite. In this paper, we propose a highly cost-effective distributed disk-based graph-processing framework, called DD-Graph that has the salient feature of both the low hardware cost and high performance.

There are two key challenges in the design of a distributed external memory based framework, that is, the expensive communication in distributed in-memory graph-processing frameworks and the high disk I/O latency that the designers of external memory based graph-processing frameworks are most concerned about. By using a small cluster to hide the latencies of the communication and disk I/O intelligently, DD-Graph overcomes the two key challenges and thus reduces the overall runtime to the computation time of the compute nodes, achieving the comparable performance with existing distributed in-memory graph-processing frameworks.

The rest of the paper is structured as follows. Section 2 introduces DD-Graph framework. Experimental evaluations are presented in Section 3. We discuss related work in Section 4 and conclude the paper in Section 5.

2. DD-GRAPH FRAMEWORK

Key components and unique features of the DD-Graph framework are detailed in the subsections that follow.

2.1 Definitions

An input graph is partitioned into P subgraphs in preprocessing phase. A graph-computing job consists of N iterations. A **task** is defined as the execution process of a subgraph in one iteration, therefore there are $T = P \times N$ tasks, ordered as T_0, T_1,$\cdots$, $T_{P\times N-1}$. A task is decomposed into three stages: 1) loading subgraph from the disk, 2) computation, and 3) communication and saving results to the disk.

In the preprocessing phase, the edge values of each subgraph are organized into a local-edge data block, P-1 remote in-edge data blocks and P-1 remote out-edge data blocks intelligently. Each remote out-edge data block includes the values of all the out-edges whose destination vertices re-

HPDC'16, May 31-June 04, 2016, Kyoto, Japan
 ISBN 978-1-4503-4314-5/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2907294.2907299

side within one of the remote subgraphs. Each remote in-edge data block includes the values of all the in-edges whose source vertices reside within one of the remote subgraphs. Since the out-edges of a given vertex are the in-edges of its neighbors, each remote out-edge data block is an exact copy of a remote in-edge data block of one of the other subgraphs. The local-edge data block is a special case because both the source vertices and destination vertices of the edges are in the same subgraph. This block-based edge-value organization method enables an efficient block-based communication model, described in the next subsection.

2.2 Architecture of DD-Graph

The DD-Graph architecture consists of a master node and M compute nodes. The master node schedules the tasks of the graph-computing job. Each compute node is responsible for managing and executing its assigned tasks. Without the loss of generality, we assume that there are three compute nodes, a graph is partitioned into six subgraphs, and a graph-computing job consists of two iterations. Thus, there are 12 tasks in total (T_0, T_1,$\cdots$, T_{11}). We describe the key components and unique features of the DD-Graph architecture as following.

Task Assignment: Tasks are assigned in order by using a hash of the task ID to select a compute node, the hash function is defined as $H(t) = t \bmod M$, where t is the ID of task t and M is the number of compute nodes. As shown in Table 1, the tasks T_0,T_3,T_6,T_9 are assigned to compute node 0, the tasks T_1,T_4,T_7,T_{10} are assigned to compute node 1, and the tasks T_2,T_5,T_8,T_{11} are assigned to compute node 2. Each compute node employs a *task queue* to store its tasks.

Table 1: Task Assignment.

Compute Node 0	Compute Node 1	Compute Node 2
T_0,T_3,T_6,T_9	T_1,T_4,T_7,T_{10}	T_2,T_5,T_8,T_{11}

Table 2: Subgraph Assignment.

Compute Node 0	Compute Node 1	Compute Node 2
S_0,S_3	S_1,S_4	S_2,S_5

Subgraph Assignment: Subgraphs are assigned in order by using a hash of the subgraph ID to select a compute node, with the hash function being defined as $H(s) = s \bmod M$, where s is the ID of subgraph s. As shown in Table 2, subgraphs S_0 and S_3 are assigned to compute node 0, subgraphs S_1 and S_4 are assigned to compute node 1, and subgraphs S_2 and S_5 are assigned to compute node 2. In order to avoid multiple copies of a subgraph, we impose a constraint condition that the number of subgraphs P is divisible by M.

Table 3: Associate Task with Subgraph.

Compute Node 0	Compute Node 1	Compute Node 2
<T_0,S_0><T_3,S_3>	<T_1,S_1><T_4,S_4>	<T_2,S_2><T_5,S_5>
<T_6,S_0><T_9,S_3>	<T_7,S_1><T_{10},S_4>	<T_8,S_2><T_{11},S_5>

Association between Task with Subgraph: Each task is associated with a subgraph by using a hash of the task ID to select a subgraph, with a hash function $H(t) = t \bmod P$, where t is the ID of task t and P is the number of the subgraphs. As shown in Table 3, for compute node 0, the tasks T_0,T_3,T_6,T_9 are associated respectively with Subgraphs S_0, S_3, S_0, S_3; for compute node 1, the tasks T_1,T_4,T_7,T_{10} are associated respectively with Subgraphs S_1, S_4, S_1, S_4; and for compute node 2, the tasks T_2,T_5,T_8,T_{11} are associated respectively with Subgraphs S_2, S_5, S_2, S_5.

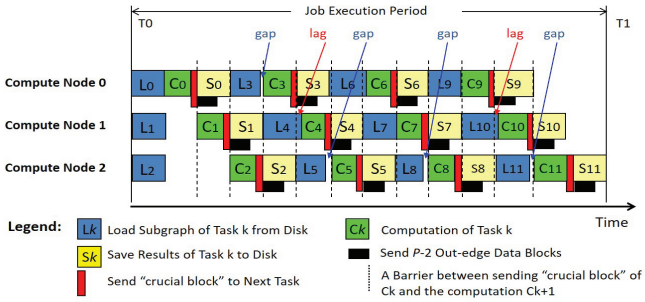


Figure 1: Job Execution Process.

Job Execution Process: As shown in Figure 1, all compute nodes start at time T0. Each compute node launches its first task from its *task queue* and loads the subgraph of that task. When the stage of loading subgraph has finished, the compute node either immediately executes the computation stage of the task t currently being launched or wait for a short time period for the arrival of the last remote out-edge data block, called the "crucial block", which is sent by the task *t-1*. The task T_0 is a special case because it is the first one. The short waiting time period indicates that the computation stage of task *t-1* has not finished. It can be eliminated by using more compute nodes. This is a trade-off between the system performance and hardware costs, as discussed in the following subsections.

In the computation stage, a user-defined **Update(v)** function is invoked for each vertex **v** in the subgraph in parallel. Inside **Update(v)**, the vertex **v** updates its state by its in-edge values and then updates its out-edge values. The in-edge values of vertex **v** were updated by the source vertices of the in-edges in the previous *P-1* tasks, and the out-edge values of vertex **v** will be used by the destination vertices of the out-edges in the subsequent *P-1* tasks.

Then, the compute node starts the block-based communication and result-saving processes simultaneously. In the block-based communication process, edge values are moved to implement the interactions between vertices, since the out-edges of a vertex are the in-edges of its neighboring vertices. The compute node sends P-1 remote out-edge data blocks to the subsequent P-1 tasks sequentially in order. Each of these P-1 tasks updates the corresponding remote in-edge data block of its subgraph by using the received remote out-edge block. In the result-saving process, the compute node saves the local-edge data block and the vertex values of the subgraph currently being executed to the disk.

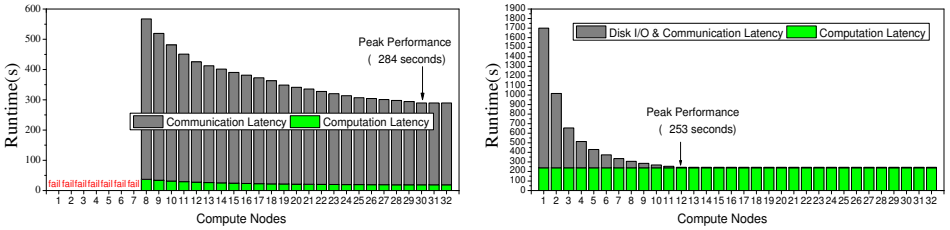
Finally, the compute node either repeats for the next task, or stops when it is demanded by the master node or the *task queue* is empty.

Number of Iterations: DD-Graph can run a fixed number of iterations by assigning N a fixed number. The graph-computing job can also proceed in an uncertain number of iterations until the convergence condition of the graph-computing job is met. In this case, a very large default value (such as 9999) of N is automatically assigned by DD-Graph.

2.3 Hardware Cost and Performance

DD-Graph overcomes two key challenges, including the high communication costs in distributed graph-processing frameworks and the high disk I/O latency in single-node external memory based graph-processing frameworks.

Minimizing Communication Cost: As illustrated in Figure 1, by using a small cluster, DD-Graph almost hides the communication latency by overlapping the communica-



(a) GPS (b) DD-Graph

Figure 2: Runtime Breakdown.

tion of each compute node with the computations of other compute nodes. Furthermore, in the communication process of each task, the compute node of the task only needs to send P-1 out-edge data blocks to the subsequent P-1 tasks sequentially in order. This communication model not only utilizes network bandwidth more efficiently but also depends weakly on the network bandwidth.

Hiding Disk I/O Latency: In order to mitigate the costly disk I/O latency [1], DD-Graph fully overlaps the processes of loading subgraph and saving results of a task with the computations of other physically distributed tasks.

By using a small cluster, the overall runtime is almost reduced to the computation time of compute nodes, achieving a comparable high-performance with Pregel-like distributed in-memory graph-processing frameworks, such as GPS [10].

2.4 Balancing Efficiency and Performance

There are *gaps* and *lags* during the execution process in DD-Graph, as shown in Figure 1. A *gap* represents an idle period between the loading subgraph stage and the computation stage of a task. For example, while L_3, the subgraph loading for the next task (Task 3), has finished, C_2, the computation of the current task (Task 2) has not finished, meaning that compute node 0 will be idle for a short period in which the computation stage of Task 3 waits for the computation stage of Task 2 to complete and the crucial block from Task 2. A *lag* signifies a short time span between two adjacent computations. For example, C_3, the computation of the current task (Task 3), has finished, but L_4, the subgraph loading for the next task (Task 4), has not finished, delaying the start of C_4. While *lags* bring extra latency between two adjacent computations and thus lengthen the overall run time, *gaps* result in the waste of computational resources and lead to efficiency loss.

More compute nodes bring more and longer *gaps* but fewer and shorter *lags*. The *lags* can be eliminated completely when the number of compute nodes is sufficiently large. In this case, DD-Graph reaches its peak performance. Inversely, fewer compute nodes result in fewer and shorter *gaps* but more and longer *lags*. DD-Graph can increase or decrease the *system scale* to tradeoff between performance and efficiency. In general, while the performance of DD-Graph increases with the *system scale*, its efficiency is inversely correlated to the *system scale*. The efficiency reaches a maximum value when *system scale*=1.

The *system scale* that achieves the peak performance depends on the specific graph algorithm since the computation time and load/store time vary from one graph algorithm to another. Even so, the peak-performance system scales are usually much smaller than those of existing distributed in-memory graph-processing frameworks while achieving the high performance of the latter, as shown in Section 3.

3. EXPERIMENTAL EVALUATION

In this section, we conduct extensive experiments to evaluate the performance of DD-Graph. Experiments are conducted on a 50-node cluster. Each node has two quad-core Intel Xeon E5620 processors with 32GB of RAM.

We implement four graph algorithms to evaluate DD-Graph: PageRank (PR), Community Detection (CD), Connected Components (CC) and RandomWalks (RW). We evaluate DD-Graph by using two real-world graph datasets and six synthetic graph datasets that are summarized in Table 4. We compare DD-Graph with two baseline frameworks. One is an up-to-date version of GPS, which is an open-source Pregel implementation from Stanford InfoLab [10]. The other is GraphChi, an open-source project from CMU [1].

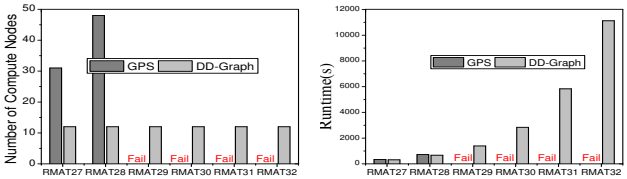
Table 4: Summary of Graph Datasets.

DataSets	Vertices	Undirected Edges	Type
Twitter-2010	41×10^6	1.4×10^9	Social Network
UK-2007-05	106×10^6	3.7×10^9	Web
RMAT27	128×10^6	2×10^9	Synthetic
RMAT28	256×10^6	4×10^9	Synthetic
RMAT29	512×10^6	8×10^9	Synthetic
RMAT30	1×10^9	16×10^9	Synthetic
RMAT31	2×10^9	32×10^9	Synthetic
RMAT32	4×10^9	64×10^9	Synthetic

3.1 Hardware Cost & Performance

In order to have a clear understanding about how DD-Graph achieves high performance and low hardware costs, experiments are conducted to investigate the runtime breakdowns of DD-Graph and GPS. Each framework runs 10 iterations of PR on the Twitter-2010 graph repeatedly, with the number of compute nodes ranging from 1 to 32. We decompose the runtime of DD-Graph into two parts: (1) computation latency and (2) disk I/O & communication latency. The runtime of GPS consists of two parts: (1) computation latency and (2) communication latency.

Experimental results, shown in Figure 2, indicate that the computation latency of DD-Graph maintains a constant value when the system scale ranges from 1 to 32. The reason for this is that the computations of the physically distributed tasks are actually executed sequentially in time by DD-Graph. Due to the parallel execution of computations, the computation latency and communication latency of GPS are reduced gradually as the system scale increases from 8 to 30. However, the runtime of GPS maintains a constant value when the system scale ranges from 30 to 32. The reason is most likely the limited scalability of GPS. Note that GPS fails to execute the graph-computing job when system scale is less than 8. However, the disk I/O & communication latency of DD-Graph is reduced significantly when the system scale ranges from 1 to 12 and reaches the peak performance at the system scale of 12. The reason is that most of the communication and disk I/O time has been overlapped when 12 compute nodes are used. Although the computation la-



(a) Compute Nodes (b) Runtime

Figure 3: Super-large Scale Graphs.

tency of DD-Graph is longer than that of GPS, the disk I/O & communication latency of DD-Graph is much shorter than the communication latency of GPS, leading to slightly shorter overall runtime of DD-Graph (i.e., 253 seconds vs. 284 seconds). Although there is a significant difference in the system scale, DD-Graph can achieve the high performance of GPS. We also repeat this experiment on CC, CD and RW respectively, and get the similar results.

3.2 Super-large Scale Graphs

We compare DD-Graph with GPS using a set of graphs, i.e., RMAT27, RMAT28, RMAT29, RMAT30, RMAT31 and RMAT32. Since the experimental results, shown in Section 3.1, indicate that DD-Graph can achieve its peak performance on 12 compute nodes for the PR graph algorithm. DD-Graph runs PR with these graphs respectively on the same 12-node cluster.

GPS first runs PR on the RMAT27 graph repeatedly, by increasing the system scale. GPS simply crashes when the system scale falls below 11, but as the system scale increases its runtime is reduced gradually and reaches the minimum value (340.6s) when 31 compute nodes are used. However, DD-Graph with only 12 compute nodes executes the same graph-computing job in 312.5 seconds. Experiments are repeated on RMAT28. Similarly, GPS reaches its peak performance (723.9s) when 48 compute nodes are used, while DD-Graph with only 12 compute nodes executes the same graph-computing job in 673.5 seconds. As shown in Figure 3(a), DD-Graph saves more compute nodes when handling RMAT28 than RMAT27. Furthermore, it can obtain slight performance improvements when handling both RMAT28 and RMAT27, as shown in Figure 3(b). These experimental results indicate that DD-Graph is more cost-effective when handling a larger graph than a smaller one.

As shown in Figure 3, GPS simply crashes when running on the 50-node cluster with RMAT29, RMAT30, RMAT31 and RMAT32, due to the out-of-memory problem. However, DD-Graph can process the RMAT29, RMAT30, RMAT31 and RMAT32 respectively on a cluster of 12 compute nodes.

3.3 Comparison with GraphChi

For fair comparison, DD-Graph is deployed on a single compute node. Each framework runs 10 iterations of PR on the graphs with different types and sizes. Experimental results indicate that DD-Graph can process the graphs of different sizes, and the runtimes of DD-Graph are similar to those of GPS when running on Twitter-2010, UK-2010-05, RMAT27, RMAT28 and RMAT29. However, GraphChi simply crashes when handling RMAT30 and RMAT31 graphs.

4. RELATED WORK

Chaos [8] scales X-Stream [9] out to multiple machines. This system reduces the disk I/O latency by using two measures. It avoids the random accesses by streaming completely unordered edge lists, aiming to improve the disk I/O performance. Within each machine, the disk I/O is partially overlapped with computation. Chaos' system performance relies heavily on the high-bandwidth networks [8], while DD-Graph has the higher communication efficiency. The extreme requirement of network in Chaos is not easily met for most small and medium-sized organizations. Unlike Chaos, DD-Graph hides almost all of the communication latency and full I/O latency by overlapping the disk I/O and communication of each compute node with the computations of other compute nodes, thus achieving the higher performance but the lower hardware cost. Furthermore, Chaos adopts an edge-centric programming model while DD-Graph adopts the vertex-centric programming model that is more user-friendly.

5. CONCLUSION

This paper proposes DD-Graph, a distributed disk-based graph-processing framework. By scheduling the tasks of a graph-computing job on a small cluster, DD-Graph is capable of processing super-large scale graphs while achieving the high performance of Pregel-like distributed in-memory graph-processing frameworks. Extensive evaluation, driven by very large-scale graph datasets, indicates that the cost-effective advantage of DD-Graph makes it notably superior to the existing distributed graph-processing frameworks.

Acknowledgment

This work is supported in part by the National High Technology Research and Development Program (863 Program) of China under Grant No.2013AA013203 and National Basic Research 973 Program of China under Grant 2011CB302301. This work is also supported by State Key Laboratory of Computer Architecture (No.CARCH201505).

6. REFERENCES

[1] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI'12*.

[2] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[3] G. Malewicz, M. H. Austern, and etc. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD'10*.

[4] J. Malicevic, A. Roy, and W. Zwaenepoel. Scale-up graph processing in the cloud: Challenges and solutions. In *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*. ACM, 2014.

[5] R. Pearce, M. Gokhale, and N. M. Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proc. SC'14*.

[6] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proc. SC'10*.

[7] R. Pearce, M. Gokhale, and N. M. Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *IPDPS'13*.

[8] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proc. ACM SOSP'15*.

[9] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proc. ACM SOSP'13*.

[10] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proc. ACM SSDBM'13*.