Efficient Graph Mining on Heterogeneous Platforms in the Cloud

Tao Zhang(✉), Weiqin Tong, Wenfeng Shen, Junjie Peng, and Zhihua Niu

School of Computer Engineering and Science, Shanghai University,
99 Shangda Road, Shanghai 200444, China
{taozhang,wqtong,wfshen,jjie.peng}@shu.edu.cn, zhniu@staff.shu.edu.cn

Abstract. In this Big Data era, many large-scale and complex graphs have been produced with the rapid growth of novel Internet applications and the new experiment data collecting methods in biological and chemistry areas. As the scale and complexity of the graph data increase explosively, it becomes urgent and challenging to develop more efficient graph processing frameworks which are capable of executing general graph algorithms efficiently. In this paper, we propose to leverage GPUs to accelerate large-scale graph mining in the cloud. To achieve good performance and scalability, we propose the graph summary method and runtime system optimization techniques for load balancing and message handling. Experiment results manifest that the prototype framework outperforms two state-of-the-art distributed frameworks GPS and GraphLab in terms of performance and scalability.

Keywords: Graph mining · GPGPU · Graph partitioning · Load balancing · Cloud computing

1 Introduction

In recent years, various graph computing frameworks [1,3–5] have been proposed for analyzing and mining large graphs especially web graphs and social graphs. Some frameworks achieve good scalability and performance by exploiting distributed computing. For instance, Stratosphere [6] is a representative graph processing framework based on the MapReduce model [7]. However, recent research has shown that graph processing in the MapReduce model is inefficient [8,9]. To improve performance, many distributed platforms adopting the vertex-centric model [5] have been proposed, including GPS [4], GraphLab [2] and PowerGraph [10]. To ensure performance, these distributed platforms require a cluster or cloud environment and good graph partitioning algorithms [1].

Previously, we proposed the gGraph [12] platform which is a non-distributed platform that can utilize both CPUs and GPUs (Graph Processing Units) efficiently in a single PC. Compared to CPUs, GPUs have higher hardware parallelism [15] and better energy efficiency [14]. However, non-distributed platforms are unable to process large-scale graphs by utilizing powerful distributed computing/cloud computing which is widely available. Therefore, in this work, we

J. Wan et al. (Eds.): CloudComp 2016, SPNCE 2016, LNICST 197, pp. 12–21, 2018.
https://doi.org/10.1007/978-3-319-69605-8_2

focus on developing methods and techniques to build an efficient distributed graph processing framework on hybrid CPU and GPU systems. Specifically, we develop these major methods and techniques: (1) A graph-summary method to optimize graph computing efficiency; (2) A runtime system for load balancing and communication reducing; (3) A distributed graph processing system architecture supporting hybrid CPU-GPU platforms in the cloud. We developed a prototype system called HGraph (that is, graph processing on hybrid CPU and GPU platforms) for evaluation. HGraph is based on MPI (Message Passing Interface), and integrates the vertex-based programming model, the BSP (Barrier Synchronous Parallel) computing model and the CUDA GPU execution model. We evaluate the performance of HGraph with both realworld and synthetic graphs in a virtual cluster on Amazon EC2 cloud. The preliminary results demonstrate that HGraph outperforms evaluated distributed platforms.

The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 presents the system overview. Section 4 presents the details of the design and implementation. The experiment methodology is shown in Sect. 5 and the result is analyzed in Sect. 6. Section 7 concludes this work.

2 Related Work

The related work can be categorized into graph processing frameworks targeting dynamic graphs and static graphs. The design and architecture of frameworks are fundamentally different depending on the type of the graph.

Realworld graphs are mostly dynamic which are evolving over time. For example, the structure of a social network is ever-changing: vertices and edges change when a user add a new friend or delete an old friend. Frameworks for dynamic graph processing generally adopt the streaming/incremental computing technique in order to handle the variation of the graph and return results in realtime or near realtime. Several work propose to take a snapshot of the graph periodically and then process it based on historical results [16,17]. The graph snapshots they process are complete graphs. In contrast, other frameworks propose to process only the changed portion of graphs in an incremental fashion [18–20]. However, not all graph algorithms can be expressed into the incremental manner, so the applications of such incremental frameworks are limited.

By taking a snapshot of a dynamic graph at a certain time, a dynamic graph can be viewed as a series of static snapshots. Most of the existing graph processing frameworks focus on dealing with static graphs (i.e. snapshots). These frameworks can be grouped into non-distributed ones and distributed ones depending on the number of computing nodes they can control. GraphChi [1], Ligra [11], gGraph [12] and Totem [13] are representative *non-distributed platforms*. The former two platforms are pure-CPU platforms. GraphChi proposed the Parallel Sliding Windows (PSW) method and the compact graph storage method to overlap the computation and I/O to improve performance. Ligra is specifically designed for shared-memory machines. Both gGraph and Totem run on hybrid CPU and GPU systems and achieve better performance and energy efficiency than pure-CPU based platforms. Anyways, non-distributed platforms cannot utilize distributed computing nodes to handle extra-scale graphs. In contrast, the performance of distributed platforms can scale up by utilizing more computing nodes in the cluster. Distributed platforms can be further classified into synchronous platforms and asynchronous platforms according to their computing model. Pregel [5] and GPS [4] are typical *distributed synchronous platforms*. Pregel and GPS adopt the vertex-centric model, in which a vertex kernel function will be executed in parallel on each vertex. GraphLab [2] and PowerGraph [10] are representative *distributed asynchronous platforms*. They follow the asynchronous computing model such that graph algorithms may converge faster. However, research showed that asynchronous execution model will reduce parallelism [12]. Therefore the selection between synchronous and asynchronous model is a tradeoff between algorithmic convergence time and performance.

3 System Overview

In this section, we discuss the design principle of the HGraph, followed by a system architecture overview. The detailed optimization techniques of HGraph will be presented in the next section.

3.1 Design Rules

The primary design goals of HGraph include good performance, scalability and programmability.

- HGraph exploits GPU computing for good performance. GPU processors are advantageous for their high throughput [21], energy efficiency [22], and memory bandwidth, and have been widely used in various application domains [15]. HGraph can benefit from GPUs' high throughput to process fine-grained computing tasks in graph processing. In addition, HGraph adopts fully in-memory computing for better performance.
- HGraph utilizes distributed computing in the cloud for good scalability. The computing resource in clouds are elastic which can scale according to users' needs. Since HGraph adopts in-memory computing, we need to ensure that there are enough nodes such that the computing resource (i.e. CPU & GPU processors) and memory resource are adequate.
- HGraph follows the vertex-centric programming model for good programmability. In this model, a specific vertex kernel function for a graph algorithm is executed in parallel on each vertex. Many existing graph processing frameworks [5,11–13] follow this model.

3.2 System Architecture Overview

The system architecture of HGraph is presented in Fig. 1. The master node consists of three major components: a graph partitioner, a task scheduler and a

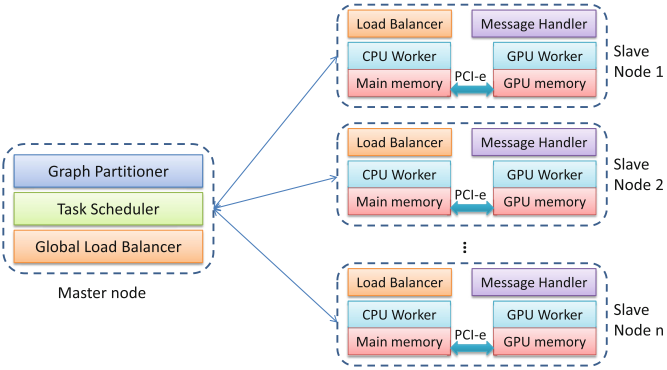

Fig. 1. System architecture of HGraph

global load balancer. The graph partitioner splits the graph into partitions and sends them to slave nodes. The task scheduler maintains a list of pending tasks and dispatches these tasks to slave nodes for execution. The global load balancer is part of the two-level load balancing unit in HGraph. The master node assigns initial load to slave nodes. Then the global load balancer can adjust the load on slave nodes if load imbalance happens during the execution.

In each slave node, there is a CPU worker and a GPU worker, respectively. The discrete GPU communicates with the host CPU through the PCI-e bus. The CPUs and GPUs inside a node work in the Bulk-Synchronous Parallel (BSP) [24] model to execute the update function in the vertex-centric programming model. However, heterogeneous processors (eg. CPUs and GPUs) may take different time for computation. As a result, completed processors need to wait for processors lagging behind before the synchronization, which degrades system performance. The local load balancer is in charge of balancing the load between the CPU and the GPU to solve such issue. The local load balancer and the global load balancer form a two-level load balancer. Finally, there is a massage handler which handles both intra-node and inter-node messages.

4 Design and Implementation

In this section, we present the methods and techniques proposed in this work. The graph summary method is introduced first, followed by the runtime system techniques for load balancing and message handling.

4.1 Graph Summary Method

In the vertex-centric model, partial or all vertices with their edges will be visit once in each iteration for many graph algorithms. Therefore the execution time is proportional to the number of vertices and edges ($O(|V|+|E|)$), and is dominated by the number of edges $|E|$ in most cases since normally $|E|$ is much bigger than

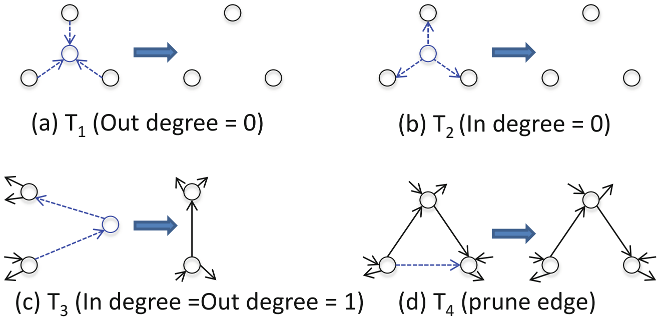

Fig. 2. Graph pruning transforms

the number of vertices $|V|$ in graphs. We define four pruning transformations T_i of graph G, as shown in Fig. 2.

T_1 is a transform that removes the vertices and their in-edges whose out-degree equals zero. T_2 is a transform that removes the vertices and their out-edges whose in-degree equals zero. T_3 is a transform that removes the vertices and their out-edges whose in-degree and out-degree both equal 1. T_4 is a transform that removes one edge from a triangle. By applying one of T_i or a series of T_i onto the G, we can get a graph summary G' of smaller size.

$$G' = T_i(G) \tag{1}$$

The selection of T_i depends on algorithms and query conditions. Queries using graph algorithms can be categorized into full queries and conditional queries:

- Full queries: using graph algorithms to identify the maximum, minimum value or all value under certain criteria. For instance, "search for the top 10 vertices with the highest PageRank", or "find out all communities in the graph".
- Partial queries: using graph algorithms to search for some solution. For instance, "search for 10 vertices with PageRank larger than 5", or "find out 10 communities whose sizes are larger than 50".

Accordingly, graph summary G' can be used in two ways:

- As the initialization data: in full query, we can use graph summary G' to initialize G to make graph algorithms converge faster [23].
- As the input for graph algorithms: in partial query, we can directly run graph algorithms on graph summary G' to get results in a shorter time.

The time for pruning vertices and edges to get graph summary is a one-time process, so the time cost can be amortized by later long-running time of iterative graph algorithms. Besides, some graph algorithms have similar algorithmic pattern such that they can share a common graph summary. Therefore, the time cost to produce graph summary can be further amortized.

4.2 Runtime System Techniques

There are two major components in the runtime system: the two-level load balancer and the message handler, as shown in Fig. 1 in Sect. 3. The local load balancer in each slave node exploits the adaptive load balancing method in gGraph [12] to balance the load between CPU processors and GPU processors inside the node. The global load balancer in HGraph is able to adjust the load (eg. number of vertices and edges) on slave nodes to balance their execution time. It calculates the load status of slave nodes based on the monitoring data and tries to migrate appropriate load from heavily loaded nodes to less loaded nodes.

We extended the message handler in gGraph for HGraph's distributed computing. In HGraph, the message handler in each slave node maintains one outbox buffer for every other slave nodes and an inbox buffer for itself. Messages to other slave nodes will be aggregated based on the slave node id and the vertex id using algorithm operators then put into the corresponding outbox buffer. The inbox buffer is used for receiving incoming messages.

5 Experiment Methodology

In this section, we elaborate the graph algorithms, graph data, and the experimental software and hardware settings.

5.1 Graph Algorithms

We use single source shortest path (SSSP), connected components (CC), and PageRank (PR) to evaluate the performance of HGraph, as shown in Table 1.

Single source shortest path finds the shortest path from a given source vertex to all connected vertices. Connected component is used to detect regions in graphs. PageRank is an algorithm proposed by Google to calculate probability distribution representing the likelihood that a web link been clicked by a random user. Their vertex functions are listed in Table 1.

Table 1. Graph Algorithms

Algorithms	Vertex function
SSSP	$v.path \leftarrow min_{e \in inEdges(v)}(e.source.path + e.weight)$
CC	$v.component \leftarrow max_{e \in edges(v)}(e.other.component)$
PR	$v.rank \leftarrow 0.15 + 0.85 \times \sum_{e \in inEdges(v)} e.source.rank$

5.2 Workloads

We use both real-world graphs and synthetic graphs in the RMAT model [25] to evaluate HGraph. The RMAT graphs are generated with parameters $(A, B, C) = (0.57, 0.19, 0.19)$ and an average degree of 16. The graphs are listed in Table 2.

Table 2. Summary of the workloads (Legend: M for million, B for billion)

Abbr.	Graph	|Vertices|	|Edges|	Direction	Type
G1	Twitter 2010	61.6M	1.5B	Undirected	Social
G2	Com-Friendster	65.6M	1.8B	Undirected	Social
G3	Uk-2007-d	106.0M	3.7B	Directed	Web
G4	RMAT29	512.0M	8.0B	Undirected	Synthetic
G5	RMAT30	1.0B	16.0B	Undirected	Synthetic

5.3 Software and Hardware Settings

We developed a system prototype named HGraph on top of MPICH2. We conducted the experiments on Amazon EC2, using 32 g2.2xlarge instances. Each g2.2xlarge instance consists of 1 Nvidia GPU, 8 vCPU, 15 GB memory and 60 GB SSD disk. Each GPU has 1536 CUDA cores and 4 GB DDR memory. We compare the performance and scalability of HGraph with two distributed frameworks GraphLab and GPS.

6 Results and Analysis

In this section, we present the comparison on performance and scalability of HGraph with GPS and GraphLab. Figure 3 compares the performance of HGraph, GPS and GraphLab running the CC, SSSP, and PR algorithm. All

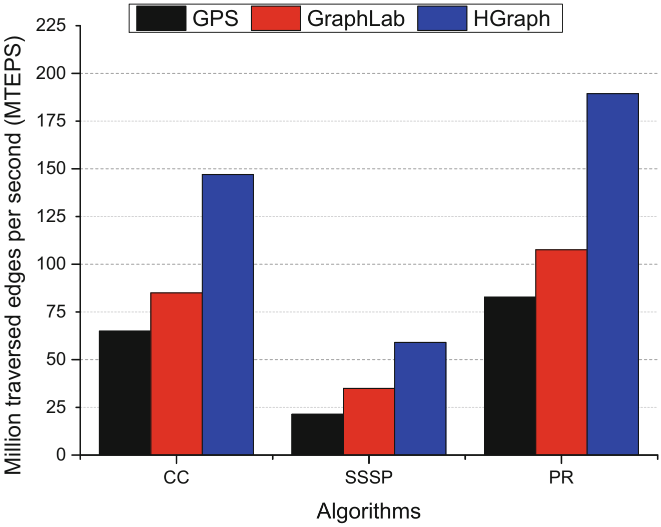

Fig. 3. Performance comparison of platforms

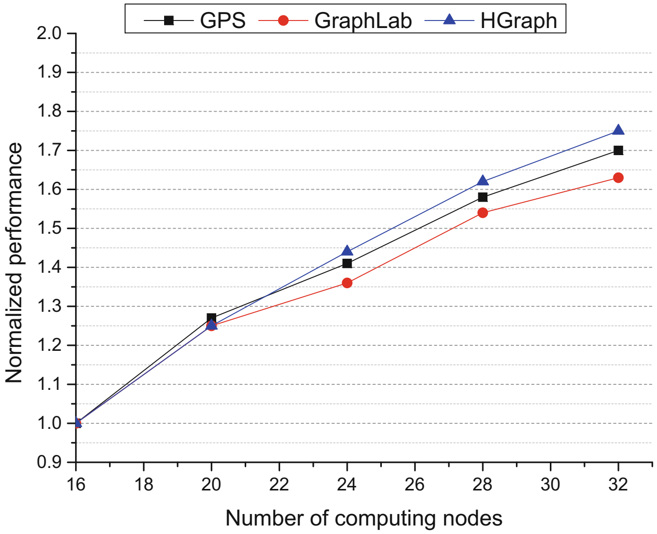

Fig. 4. Scalability comparison of platforms

three platforms are distributed but only HGraph can utilize GPUs in computing nodes and gain additional computing power. The result is the average performance in million traversed edges in one second (MTEPS) on all graphs. In general, platforms achieve better performance in graph analytical algorithms (CC & PR) than in the graph traversal algorithm (SSSP) since CC & PR have higher parallelism than SSSP. HGraph outperforms GPS and GraphLab for two reasons: (1)the graph summary method and the runtime system optimizations; (2) the ability to utilize GPUs for additional power.

Figure 4 compares the scalability of three platforms by increasing the number of computing nodes from 16 to 32 at a step of 4 machines, and calculating the normalized performance. All platforms exhibit significant scalability. HGraph achieves the best scalability while GraphLab achieves the lowest scalability. Adding one or more computing nodes increases the resource including processors, memory and disk I/O bandwidth, and reduces the partitioned workload on each computing node. However, more computing nodes also cause the graph to be split into more partitions, potentially increasing communication messages. HGraph implements the message aggregation technique therefore it is less affected by the increased communications, hence the better scalability.

7 Conclusion

This paper introduces a general, distributed graph processing platform named HGraph which can process large-scale graphs very efficiently by utilizing both CPUs and GPUs in distributed cloud environment. HGraph exploits a graph summary method and runtime system optimization techniques for load balancing and message handling. The experiments show that HGraph outperform two state-of-the-art distributed platforms GPS and GraphLab in terms of performance and scalability.

Acknowledgment. This research is supported by Young Teachers Program of Shanghai Colleges and Universities under grant No. ZZSD15072, Natural Science Foundation of Shanghai under grant No. 16ZR1411200, and Shanghai Innovation Action Plan Project under grant No. 16511101200.

References

1. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pp. 31–46 (2012)
2. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
3. Shao, B., Wang, H., Li, Y.: Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 505–516. ACM (2013)
4. Salihoglu, S., Widom, J.: GPS: a graph processing system. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management, p. 22. ACM (2013)
5. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135–146. ACM (2010)
6. Warneke, D., Kao, O.: Nephele: efficient parallel data processing in the cloud. In: Proceedings of the 2nd Workshop on Many-task Computing on Grids and Supercomputers, p. 8. ACM (2009)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. Guo, Y., Biczak, M., Varbanescu, A.L., Iosup, A., Martella, C., Willke, T.L.: How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 395–404. IEEE (2014)
9. Pan, X.: A comparative evaluation of open-source graph processing platforms. In: 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 325–330. IEEE (2016)
10. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012), pp. 17–30 (2012)
11. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. ACM SIGPLAN Not. **48**(8), 135–146 (2013). ACM
12. Zhang, T., Zhang, J., Shu, W., Wu, M.Y., Liang, X.: Efficient graph computation on hybrid CPU and GPU systems. J. Supercomput. **71**(4), 1563–1586 (2015)

13. Gharaibeh, A., Reza, T., Santos-Neto, E., Costa, L.B., Sallinen, S., Ripeanu, M.: Efficient large-scale graph processing on hybrid CPU and GPU systems (2013). arxiv preprint arXiv:1312.3018
14. Zhang, T., Jing, N., Jiang, K., Shu, W., Wu, M.Y., Liang, X.: Buddy SM: sharing pipeline front-end for improved energy efficiency in GPGPUs. ACM Trans. Archit. Code Optim. (TACO) **12**(2), 1–23 (2015). Article no. 16
15. Zhang, T., Shu, W., Wu, M.Y.: CUIRRE: an open-source library for load balancing and characterizing irregular applications on GPUs. J. Parallel Distrib. Comput. **74**(10), 2951–2966 (2014)
16. Iyer, A.P., Li, L.E., Das, T., Stoica, I.: Time-evolving graph processing at scale. In: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, p. 5. ACM (2016)
17. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Chen, E.: Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems, pp. 85–98. ACM (2012)
18. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 439–455. ACM (2013)
19. Wickramaarachchi, C., Chelmis, C., Prasanna, V.K.: Empowering fast incremental computation over large scale dynamic graphs. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), pp. 1166–1171. IEEE (2015)
20. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. IEEE Trans. Parallel Distrib. Syst. **25**(8), 2091–2100 (2014)
21. Han, S., Lei, Z., Shen, W., Chen, S., Zhang, H., Zhang, T., Xu, B.: An approach to improving the performance of CUDA in virtual environment. In: 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 585–590. IEEE (2016)
22. Jing, N., Jiang, L., Zhang, T., Li, C., Fan, F., Liang, X.: Energy-efficient eDRAM-based on-chip storage architecture for GPGPUs. IEEE Trans. Comput. **65**(1), 122–135 (2016)
23. Wang, K., Xu, G., Su, Z., Liu, Y.D.: GraphQ: graph query processing with abstraction refinement scalable and programmable analytics over very large graphs on a single PC. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp. 387–401 (2015)
24. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
25. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: a recursive model for graph mining. SDM **4**, 442–446 (2004)