Pimiento: A Vertex-Centric Graph-Processing Framework on a Single Machine

Jianqiang Huang[1,2], Wei Qin[1], Xiaoying Wang[2], and Wenguang Chen[1,2](✉)

[1] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
{hjq16,tanw16}@mails.tsinghua.edu.cn, cwg@tsinghua.edu.cn
[2] Department of Computer Technology and Applications, Qinghai University, Xining 810016, China
Wangxiaofu163@163.com

Abstract. Here, we describe a method for handling large graphs with data sizes exceeding memory capacity using minimal hardware resources. This method (called Pimiento) is a vertex-centric graph-processing framework on a single machine and represents a semi-external graph-computing system, where all vertices are stored in memory, and all edges are stored externally in compressed sparse row data-storage format. Pimiento uses a multi-core CPU, memory, and multi-threaded data preprocessing to optimize disk I/O in order to reduce random-access overhead in the graph-algorithm implementation process. An on-the-fly update-accumulated mechanism was designed to reduce the time that the graph algorithm accesses disks during execution. Our experiments compared external this method with other graph-processing systems, including GraphChi, X-Stream, and FlashGraph, revealing that Pimiento achieved $7.5\times, 4\times, 1.6\times$ better performance on large real-world graphs and synthetic graphs in the same experimental environment.

Keywords: Vertex-centric · Graph processing · Semi-external · Passing message · Asynchronous update accumulation

1 Introduction

With the rapid development of the internet and the big-data era, there is a need to analyze large volumes of data. As an abstract data structure, graphs are used by many applications to represent large-scale data in real scenarios, and graph data structures are used to describe the relationships among data, such as mining relationships in social networks, goods recommendations in e-commerce systems, and analysis of the impact of traffic accidents on road networks. Additionally, many types of unstructured data are often transformed into graphs for post-processing and analysis. Research into large-scale graph-processing has increased in both academia and industry, and recently, numerous systems and state-of-the

© Springer Nature Switzerland AG 2020
S. Wen et al. (Eds.): ICA3PP 2019, LNCS 11945, pp. 42–56, 2020.
https://doi.org/10.1007/978-3-030-38961-1_5

art techniques for graph processing have emerged, including distributed systems and heterogeneous systems. Such systems present new computing models or highlight the design of high-performance runtime systems used to adapt to the features of graph data, such as its large scale, ability to dynamically change, and its high efficiency when processing big graph data.

Examples of these systems include distributed graph-computing systems, such as pregel [1], GraphLab [2], PowerGraph [3], and Gemini [4], which can theoretically deal with any large-scale graph data by deploying clusters with good extensibility and computational efficiency; however, there remain problems, including maintenance of load balance between nodes and communication latency.

Other systems include single graph computing system, such as GraphChi [5], X-Stream [6], FlashGraph [12], GridGraph [8], and other external graph-processing systems [7,9–11,13,14,22], which can reduce random disk-read and disk-write operations, avoid high communication overhead, and use parallelization technology to fully exploit multi-core computing resources to address large-scale graph data. Compared with distributed systems, these exhibit lower hardware cost and power consumption.

GraphChi is a single graph-computing system using a vertex-centric calculation model and multi-threaded parallel computing to improve computing performance. It utilizes parallel sliding-window (PSW) [5] technology to reduce random access to the disk and supports asynchronous computations. GraphChi processes graphs in three stages: (1) loading graph data from the disk to memory, (2) updating the values of vertices and edges, and (3) writing updates to disk.

GraphChi exhibits good platform usability and computing performance; however, its preprocessing requires sorting of the source vertex of the edges, which is costly. Moreover, computing processes and disk I/O access are executed in serial, and the parallelism between disk I/O and the CPU is not fully utilized to overlap computing and I/O in order to further improve computing performance. By contrast, X-Stream uses an edge-centric computing model, where all states are stored in the vertex.

To address these issues, we propose Pimiento, a vertex-centric graph-processing framework that combines asynchronization with efficiency. The outline of our paper is as follows: Sect. 2 introduces disk-based graph-computation challenges, describes system design and implementation in Sect. 3, Sect. 4 describes evaluation of Pimiento on large problems (graphs with billions of edges) using a set of algorithms, such as single source shortest path (SSSP), PageRank, and breadth-first search (BFS).

The main contributions of this paper are as follows:

- We describe the use of a vertex-centric computing model with effective graph-storage structure that adopts an innovative asynchronous update-accumulation mechanism. This enables update and repeat visits to any vertex to occur in memory in order to avoid a large number of random I/O and repeat I/O operations generated by frequent updates and reads of disk data.

– Pimiento implements a semi-external asynchronous graph-processing framework to maximize on-the-fly updates via thread optimization of computing and I/O, thereby reduced access to I/O data.
– Our evaluation showed that Pimiento outperformed current state-of-the-art techniques.

2 Disk-Based Graph Computation

A graph is a data structure that describes the complex relationship between data and comprises vertices and edges usually expressed as $G = (V, E)$, where the vertex set, V, represents an object or entity, and the edge set, E, represents the relationship between objects or entities. Each vertex $v \in V$ will have a vertex value. Given a directed edge from vertex u to vertex v, $e = (u, v)$, e is the in-edges of v and the out-edges of u, where u represents the in-vertex of v, and v represents the out-vertex of u.

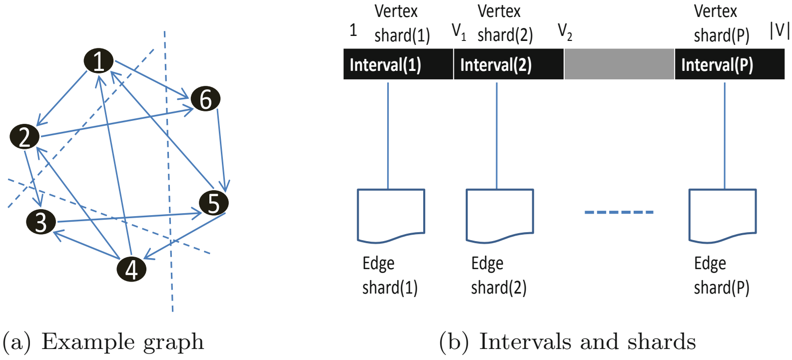
In a vertex-centric calculation model for iterative calculations, the value of each update vertex usually involves only the input vertex value. Once a vertex value is updated, a new message is sent to the output side, and the value of the output side is updated. This dynamic update of the iterative process is terminated when a convergence condition is satisfied. As framework [5] shows, a vertex-centric calculation model can address a broad range of problems. The method proposed in this paper is based on asynchronous calculations using a vertex-centric value-calculation model. Combined with the on-the-fly accumulation of the update mechanism, it promotes an effective graph-storage and calculation models. Based on the effective management of graph data, it can minimize disk data traffic and make full use of the parallel update of memory and CPU resources in order to improve computational efficiency.

2.1 Maintaining Specification Integrity

We divided vertex set V of graph into P intersecting intervals (see Fig. 1(b)). Each interval correlates with a shard that contains information needed to update the vertex calculation. As a result of the asynchrony of cumulative iterative computations, the graph partition has little effect on performance. This method only supports hash or range partitions based on a graph vertex number.

The system described by Pearce et al. [19] uses a CSR storage format to store the graph on a disk and is equivalent to storing the graph as an adjacency table, where the edges in each edge shard are sorted according to a source vertex. We call these edge data, which are stored continuously in contiguous blocks on the disk.

Suppose that the vertex set in Fig. 1(a) is divided into three intervals (interval1 = [1, 2], interval2 = [3, 4], and interval3 = [5, 6]), each of which is associated with a shard, including the edge-and vertex shards. All Vertex shards will cascade into a vertex table in order to initialize the vertex information, and all edge



(a) Example graph

(b) Intervals and shards

Fig. 1. Intervals and shards in the graph.

shards will cascade into an edge data stream in order to flow updates to the vertex information.

This graph-storage structure addresses the following three problems:

- To improve the parallelism of single-machine graph calculation, the graph-storage structure of the shard is used to render each executing thread responsible for one or more shards for parallel calculation;
- Because random access is more than an order of magnitude slower than sequential access to a disk, and given that the number of vertices in real-world graph data is smaller than the number of edges, we used memory for constant iterative updates of vertex data and secondary storage for edge lists in order to make full use of the random read-write capability of memory and the large capacity of secondary storage;
- To avoid secondary storage of random I/O, we organized edge data to ensure that access to graph data involves sequential I/O.

2.2 Computational Model

In incremental iterative calculations, graph data include read-only data by constantly updating vertex value V, as the vertex value of the cumulative value ΔV. We found that ΔV is involved in the update of adjacent vertices and will usually be accessed many times. I/O represents a bottleneck to disk-based methods, and in order to avoid frequent updates and reads of disk V and ΔV, thereby causing repeated random I/O and I/O, read-only edge data are detached from the variable-peak value of V and ΔV, and the read-only edge data are continuously stored on the edge shard disk.

We combined the cumulative iterative computations and the cache of all of the vertices values for V and ΔV into the memory. Because the space occupied by vertices values V and ΔV are less than the space occupied by the edge data, the memory capacity of the modern computer can meet the requirements. Pimiento uses flow calculation, and the space occupied by the edge list in memory

is dynamically balanced and controllable, which also proves the desirability of caching vertex data into memory. Due to the cumulative nature of the algorithm, the updates and access to peak value V and ΔV can be performed in memory. At this point, updating each interval requires only one sequential scan of the corresponding read-only edge list to minimize the I/O overhead of graph data access.

This paper is based on the traditional incremental iteration theory [15] and presents a graph-computing model in a parallel environment for application for stand-alone large graph data processing. The parallel-computing model is adopted in the framework of general graph computing, where each execution thread is responsible for one or more shards, as well as each subdivision, including the vertex shard and a corresponding edge shard. Additionally, smaller vertex shards are loaded into memory to support frequent updates, and larger edge shards are placed on the disk to save memory.

The computing framework of the diagram is shown in Fig. 2. During the implementation process of the iterative calculation, each execution thread reads the edge information sequentially from disk and updates the neighbor vertex state based on the state of vertices V and ΔV in the local Vertex shard. The communication between threads involves passing ΔV. There are two main overheads in this model: I/O overhead for reading graph data from the disk and the overhead of interthread communication. This computing model uses cumulative iterative computation to greatly reduce these two overhead issues.

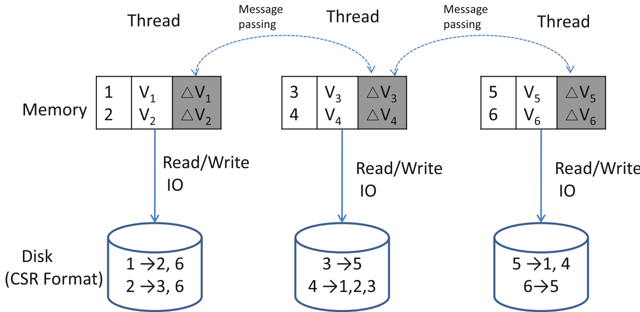


Fig. 2. Memory and secondary storage in the graph

2.3 Update Scheme

Algorithm 1 describes the implementation of the cumulative iterative-computing model in a single-machine parallel-computing environment. First, edge data is sequentially read for any vertex i, from edge shard data from the disk, and the information record of this vertex, i (V_i and ΔV_i), in the memory vertex shard is positioned according to the source vertex number of the edge data. When the vertex, i, edge data is loaded into memory, the algorithm determines whether

the vertex information is a valid change (i.e., whether ΔV_i indicates 0) for the effective information ($\Delta V_i \neq 0$). First Algorithm 1: pseudo-code of the vertex update function for weighted PageRank.

Algorithm 1 :Pseudo-code of the vertex update function for weighted PageRank.

Input: All intervals vertex-shards and edge-shards of graph G, optional initialization data.
Output: Desired output results.

```
 1: function UPDATE(vertex)
 2:     Initialize(vertex-shards);
 3:     repeat
 4:         v[i] ← read values of out-edges of vertex i ;
 5:         vertex.value ← f(v[i]) ;
 6:         if Δf(v[i]) ≠ 0 then  f(v[i]) ← Δf(v[i]) + f(v[i]) ;
 7:             for each edge of vertex do
 8:                 edge.value ← f(vertex.value, edge.value));
 9:                 Δf(v[i]) ← 0 ;
10:             end for
11:         end if
12:     until
13:     PassingMessage(vertex) ;
14:     remove outgoing edges of i
15: end function
```

Accumulate ΔV_i to vertex i and perform an update operation to use the update of ΔV_j of the neighbor vertex, j, followed by resetting the change of information in vertex i. When the operation on vertex i is completed, the edge data of vertex i is deleted from memory to free memory space for other uncomputed vertex edge data. This activity is repeated until the algorithm converges.

Table 1. Notations of a graph

Notation	Meaning
G	A graph G = (V, E)
V	Vertices in G
E	Edges in G
n	Number of vertices in G, n = [V]
m	Number of edges in G, m = [E]
P	Number of intervals
B_a	Size of a vertex attribute in bytes
B_v	Size of a vertex id in bytes
B_e	Size of an edge in bytes
B_M	Size of available memory budget in bytes
B	Size of a disk block accessed by an I/O unit

2.4 Analysis of the I/O Costs

During an iteration, GraphChi [5] processes each shard in three steps: (1) load the sub-graph from the disk; (2) update the vertex and edge values; and (3) write the updated values to the disk. In steps 1 and 3, each vertex is loaded and written back to the disk once, and the nB_v data volume is read and written. For each edge data, in the worst case, each edge is accessed twice (once in each direction). The amount of data $2m(B_v + B_e)$ will be read in step 1, the updated edge value will be calculated in step 2, and the amount of data $2m(B_v + B_e)$ will also be written in step 3. During the entire calculation, the total amount of data in GraphChi read and written is $2m(B_v + B_e)+nB_v$. During each iteration, PSW [5] generates P^2 random reads and writes, whereas in during the entire calculation process, the number of I/O read and write events for the PSW is $(2m(B_v + B_e)+nB_v)/B+P^2$, Table 1 shows the Notations of a graph.

In X-Stream [6], an iteration is divided into: (1) a mixed scatter/shuffle phase and (2) a gather phase. In phase 1, the X-Stream loads all vertex and edge data, updates each edge, and writes the updated edge data back to disk. Because the edge data after update are used to pass values between adjacent vertices, we assume that the size of an updated piece of edge data is Be; therefore, for phase 1, the amount of data read is $nB_v + mB_e$, and the amount of data written is mB_e. In phase 2, the X-Stream loads all updated edge data and updates each vertex; therefore, for phase 2, the amount of data read is nB_v and the amount of data written is nB_v. Therefore, for an iterative-calculation process, the total amount of data read by X-Stream is $(B_v + B_e)m + nB_v$, the total data amount written is $nB_v + mB_e$, the number of I/O reads is $(m(B_v + B_e) + nB_v)/B$, and the number of I/O writes is $nB_v/B + mB_v log^{P/B}_{Bm/B}$.

In FlashGraph [12], during the entire computation process, the number of I/O reads by Pimiento is $(mB_e + nPB_v)/B$, and the number of I/O writes is nB_v/B.

In Pimiento, the entire computation process loads all of the vertex shares once. During each iteration, all edge shares are loaded from disk in turn, and the entire computation process requires reading the amount of data $(mB_e + nB_a)$. After the computation, the vertex data value will be written back to disk, and the amount of data in nB_v needs to be written. Note that the edge shard is read-only. To analyze the I/O cost, we use B to represent the size of the disk block accessed by an I/O unit. According to a previous report, B is 1MB on the SSD. During the entire computation process, the number of I/O reads by Pimiento is $(mB_e + nB_a)/B$, and the number of I/O writes is nB_a/B.

3 System Design and Implementation

Based on the asynchronous incremental-update model, we implemented the Pimiento system with C++. Pimiento divides each graph-processing task into three steps:

- Graph data shard and vertex information in memory are initialized;
- Stream-load edge data into memory, update vertex information, and clear edge data in order to free memory;
- Write the final result in memory back to disk.

Optimization techniques implemented in this paper include: I/O thread optimization, memory resource monitoring, and automatic switching of memory-external memory computing.

3.1 *I/O* Thread Optimization

Pimiento initiates parallel processing by executing threads that need to read edge data on the edge shard before they can perform subsequent vertex updates, which results in a lot of I/O. Because there is no synchronization between execution threads, computation and update speeds are very fast. However, it is often necessary to wait for the end of the I/O operation; therefore, I/O represents the Pimiento performance bottleneck.

A thread execution includes an I/O operation and an update operation. The I/O operation loads edge data into memory, and the update operation updates the vertex using edge data. However, this binds the I/O operation to the update operation in a thread of execution. In this case, I/O operations and update operations are synchronized more frequently, resulting in lower I/O throughput and CPU-resource utilization.

To address these problems, Pimiento separates the I/O operation from the update operation, creating multiple update threads responsible for each vertex-update operation while creating multiple I/O threads responsible for loading edge data into memory, thereby more reasonably allocating I/O and computing resources. However, if there are too many I/O threads relative to update threads, there will be too much cache data, and the update thread will not be able to execute, which will cause the cache to rapidly expand and fill memory. If the I/O thread is too small relative to the update thread, the update thread will execute too quickly while the I/O thread will be too small to keep up with the influx of data, resulting in an idle update thread while it waits for I/O.

To avoid these situations, Pimiento allows users to set the I/O- and update thread allocations according to resource and application features in order to use a memory monitoring strategy to ensure balance between the update and I/O threads to maintain saturation of I/O and CPU resources and maximize system performance.

3.2 Memory Resource Monitoring

In Pimiento, the I/O thread reads edge data and caches it in memory while and the update thread digests the edge data to update the graph vertex state, after which memory is freed when graph edge data is used. Because the I/O thread executes in parallel with the update thread, the I/O operation is not controlled by the update thread, which could result in a mismatch between the throughput

of the graph edge data in during update thread processing and throughput of the graph edge data during I/O thread reading. If I/O throughput is too fast, this will result in increased caching of edge data loaded from disk into memory, which will eventually lead to memory overflow. If I/O throughput is too slow, this will result in the update thread remaining in a waiting state, leading to CPU-resource waste.

To address this problem, Pimiento uses a memory resource-monitoring thread to monitor memory usage. When memory for cached data is running low, the monitoring thread signals individual I/O threads to block I/O threads to prevent edge data loading in order to wait for the update thread to process the edge data and release memory. When the monitoring thread detects that memory overflow is no longer a possibility, it signals the individual I/O threads to continue loading edge data. The memory resource-monitoring strategy increases Pimiento memory efficiency, maximizes memory utilization to improve computing speed, and avoids memory overflow. The memory monitoring thread perfectly coordinates the update thread with the I/O thread, making the system more robust and coordinated while performing parallel computations and disk I/O operations.

4 Experimental Evaluation

We implemented and evaluated a wide range of applications in order to demonstrate the applicability of Pimiento to multi-domain problems. Despite the restrictive external memory setting, Pimiento retains the expressivity of other external graph-processing frameworks.

4.1 Test Setup

All experiments used a commercial server equipped with an e5-2670@v3 processor, which has two sockets running at 2.3 GHz, 32 MB L3 cache, with 12 cores per socket, and a disabled CPU hyper-threading feature. The commercial server was equipped with 32 Gbyte of memory and 1 Tbyte of disk (SSD), and the operating system was 64-bit Ubuntu 14.04 LTS. We evaluated Pimiento using the applications described in Section and analyzed its performance on a selection of large graphs (Table 2).

Table 2. Real-world and synthetic graphs data used in the experiments

Dataset	Twitter [16]	UK-2007 [17]	Rmat27 [18]
Vertex num	41.6M	134M	128M
Edge num	1.5B	5.5B	2B
Avg deg	35.3	41.2	16
Max outdeg	770K	22.4K	123K
Size	25 GB	93 GB	32 GB

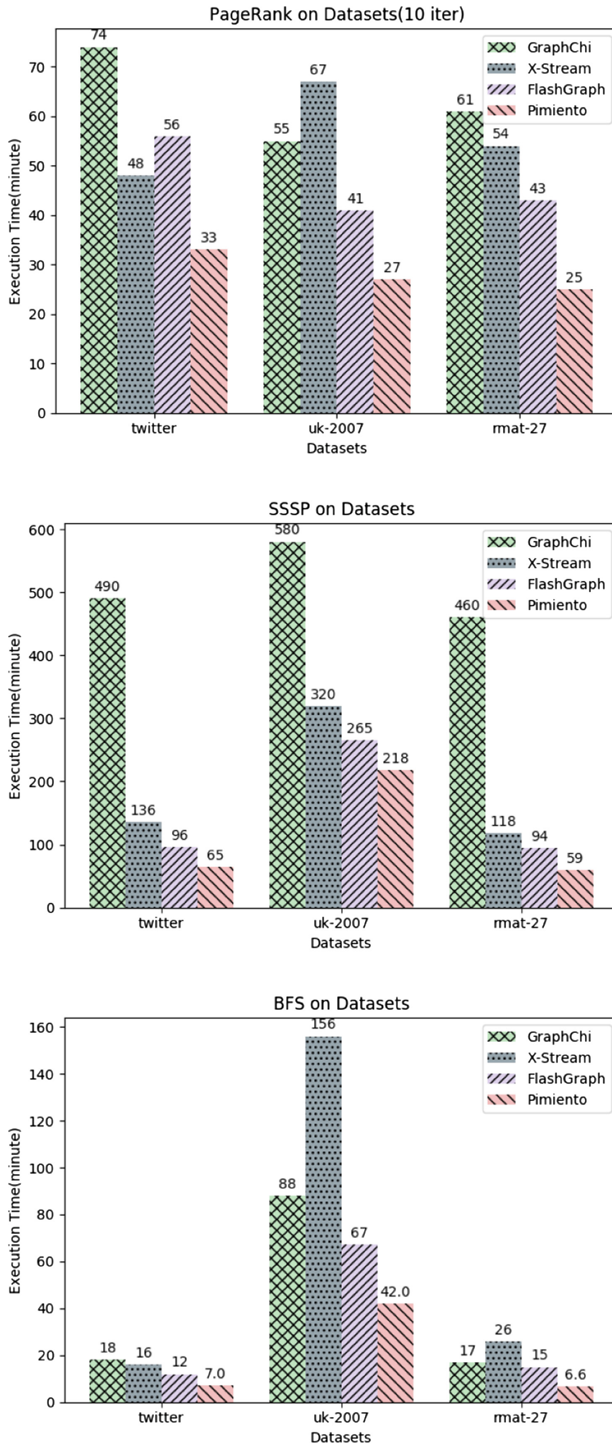


Fig. 3. Comparison of execution time when performing PageRank, SSSP and BFS over different data sets.

4.2 Comparison with Other Systems

4.2.1 Propagation-Based Algorithms

First, we evaluated graph-propagation-based traversal, such as that using BFS and SSSP. Figure 3 shows that Pimiento performed better on SSD than GraphChi and X-Stream. Compared with GraphChi, X-Stream, and FlashGraph on Twitter, Uk-2007, and Rmat27, respectively, Pimiento was 1.6 times to 7.5 times faster. There are mainly two reasons for the acceleration:

- Pimiento reads edge data sequentially from disk, thereby reducing random access to the disk
- Pimiento can reduce the amount of data written back to disk, effectively avoiding a data race.

4.2.2 Iteration-Based Algorithms

We then evaluated graph iteration-based algorithms, such as PageRank, and confirmed that PageRank is representative of a cumulative algorithm. When computing a PageRank value, each vertex should first collect all values from its source vertices in order to compute a sum. Pimiento uses a vertex-centric on-the-fly update model.

We compared four systems: Pimiento, GraphChi, X-Stream, and FlashGraph. In each iteration, the graph-processing system computed the new PageRank value for each vertex and selects the largest one. The iteration stops when the maximum PageRank value reaches a stable state (i.e., when the maximum change in PageRank value between iterations is less than the threshold value, computing is assumed to have converged and ends).

As shown in Fig. 3, Pimiento performed better on different data sets than GraphChi, X-Stream, and FlashGraph. Because Pimiento uses sequential disk access, it is multi-fold faster than GraphChi and X-Stream. Specifically, Pimiento is 2.3 times faster than GraphChi and 1.5 times faster than X-Stream on a Twitter dataset. The primary reason for this is that values of all vertices are sent to destination vertices along outer edges for cumulative updates, and there is no need to write the values of destination vertices back to disk. To evaluate the improved performance of Pimiento, we analyzed the total amount of I/O performed by the BFS, SSSP, and PageRank algorithms on different graphs (see Fig. 4). Specifically, compared with GraphChi, S-Stream, and FlashGraph, the I/O-data volume of Twitter, Uk-2007, and Rmat27 was reduced by a range of 30% to 98%, because the status values of all vertices were updated instantly, precluding the need to write the vertex state back to disk.

4.3 Optimization of the Update- and I/O Thread Proportions

When using SSD, we open multiple I/O threads in order to increase the storage capacity of data reading and computational efficiency. To explore the effect of

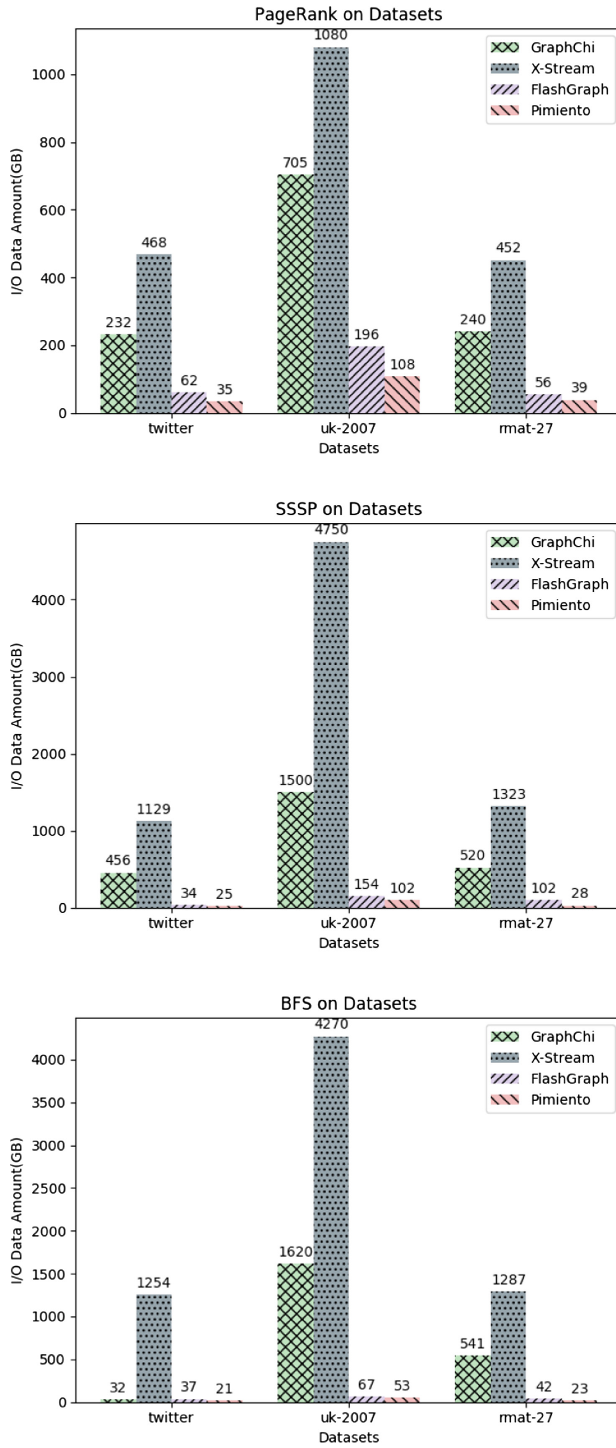


Fig. 4. Comparison of overall I/O data amount when performing PageRank, SSSP and BFS over different data sets.

the update- and I/O thread number selection on the performance of Pimiento, we compared the convergence speed of Pimiento in executing the iteration algorithm under different proportions of update and I/O threads. Figure 5 shows the average time for PageRank to converge relative to Pimiento, revealing that the convergence speed first increased and then decreased after peaking at a proportion of 4:1.

Our analyses showed that when the I/O thread was busier that the update thread, too much cache-structure data would require processing, precluding execution of the update thread. However, if the amount of data going to the I/O thread was less than that to the update thread, the update thread would execute too rapidly while the I/O would need to starve in order to maintain pace with the data input. These two situations would result in the output described in Fig. 5, which should be avoided.

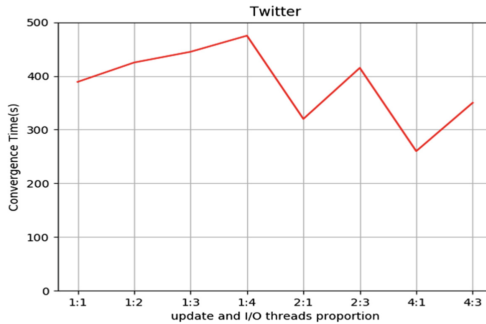


Fig. 5. Update and I/O threads proportion

5 Related Work

Here, we proposed improvements in single-computer-processing power and storage capacity using a graph-processing model. Such systems demonstrate adequate graph-processing performance, and compared with distributed systems, their obvious advantages include low hardware cost and low power consumption.

TurboGraph [9] makes full use of multi-core concurrency and the I/O performance of Flash SSD [20] to parallelize CPU processing and I/O processing in order to support rapid graph data storage. VENUS [14] is a point-centric streamlining graph-processing model that introduces a more efficient model for storing and accessing disk graph data using a cache strategy. FlashGraph [12] is a single-machine graph-processing system that can handle trillions of nodes on a solid-state hard-disk array while providing a dynamic load balancer to solve CPU-idle

results from uneven computing tasks. GridGraph [21] supports selective scheduling, which can greatly reduce I/O and improve computing performance in algorithms, such as BFS and weakly connected components. NXgraph [13] provides three update strategies: (1) sort by the target vertex of each sub-shard edge; (2) based on the size of the graph and the available memory resources, the fastest execution strategy for different graph problems is adaptively selected to take full advantage of memory space and reduce data transmission; and (3) to solve the problem of large graphs fully loaded into memory, a previous study described the design of a disk-based single graph-processing platform using MMap [10] in Linux memory management. MMap maps a file or other pair to memory, where a process can access the file just as it accesses a normal memory without using operations, such as *read*() and *write*().

6 Conclusions

There currently numerous studies focused on addressing large graph-processing problems using high-performance single-server systems. The existing single-server graph-processing system has limitations, including poor locality, heavy synchronization cost, and frequent I/O access. Our study compared out-of-core graph-computing systems, including GraphChi, X-Stream, and FlashGraph, with Pimiento, revealing that Pimiento achieved $7.5\times, 4\times, 1.6\times$ better performance on large real-world graphs and synthetic graphs in the same experimental environment.

Acknowlegements. This paper is partially supported by "QingHai Province High-end Innovative Thousand Talents Program-Leading Talents", The National Natural Science Foundation of China (No. 61762074, No.61962051), The Open Project of State Key Laboratory of Plateau Ecology and Agriculture, Qinghai University (No. 2020-ZZ-03), and National Natural Science Foundation of Qinghai Province (No. 2019-ZJ-7034).

References

1. Malewicz, G., et al.: Pregel: a system for large scale graph processing. In: Proceedings of the 2010 International Conference on Management of Data, SIGMOD 2010, pp. 135–146 (2010)
2. Low, Y., Bickson, D., Gonzalez, J., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. In: Proceedings of the VLDB Endowment, pp. 716–727 (2012)
3. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D.: PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI 2012, pp. 17–30 (2012)
4. Zhu, X., Chen, W., Zheng, W., Ma, X.: Gemini: a computation-centric distributed graph processing system. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, pp. 301–316 (2016)

5. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI 2012, pp. 31–46 (2012)
6. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-stream: edge-centric graph processing using streaming partitions. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 472–488 (2013)
7. Shao, Z., He, J., Lv, H., Jin, H.: FOG: a fast out-of-core graph processing framework. Int. J. Parallel Prog. **45**(6), 1259–1272 (2017)
8. Zhu, X.W., Han, W.T., Chen, W.G.: Grid graph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: Proceedings of the 2015 USENIX Conference on USENIX Annual Technical Conference, pp. 375–386 (2015)
9. Han, W.-S., et al.: TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 77–85 (2013)
10. Lin, Z., Kahng, M., Sabrin, K.M., Chau, D.H.P., Lee, H., Kang, U.: Mmap: fast billion-scale graph computation on a PC via memory mapping. In: IEEE International Conference on Big Data, IEEE, pp. 159–164 (2014)
11. Yuan, P., Zhang, W., Xie, C., Jin, H., Liu, L., Lee, K.: Fast iterative graph computation: a path centric approach. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 401–412. IEEE Computer Society (2014)
12. Zheng, D., Mhembere, D., Burns, R., Vogelstein, J., Priebe, C.E., Szalay, A.S.: FlashGraph: processing billion-node graphs on an array of commodity SSDs. In:13th USENIX Conference on File and Storage Technologies (FAST 2015) USENIX Association, pp. 45–58 (2015)
13. Chi, Y., Dai, G., Wang, Y., Sun, G., Li, G., Yang, H.: NXgraph: an efficient graph processing system on a single machine. In: Proceedings of the 32nd International Conference on Data Engineering, ICDE 2016, pp. 409–420 (2016)
14. Cheng, J., Liu, Q., Li, Z., Fan, W., Lui, J.C.S., He, C.: VENUS: vertex-centric streamlined graph computation on a single PC. In: Proceedings of the 31nd International Conference on Data Engineering, ICDE 2015, pp. 1131–1142 (2015)
15. 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)
16. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media? In: Proceedings of the 19th International Conference on World Wide Web, pp. 591–600 (2010)
17. Boldi, P., Santini, M., Vigna, S.: A large time-aware web graph. SIGIR Forum **42**(1), 78–83 (2008)
18. The graph 500 list (2014). http://www.graph500.org/
19. Pearce, R., Gokhale, M., Amato, N.: Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In: SuperComputing (2010)
20. Badam, A., Pai, V.S.: SSDAlloc: hybrid SSD/RAM memory management made easy. In: Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation. USENIX Association, p. 16 (2011)
21. Zhu, X., Han, W., Chen, W.: GridGraph: largescale graph processing on a single machine using 2-level hierarchical partitioning. Proceedings of the 2015 USENIX Annual Technical Conference, pp. 375–386 (2015)
22. Vora, K., Xu, G., Gupta, R.: Load the edges you need: a generic I/O optimization for disk-based graph processing. In: Proceedings of the 2016 USENIX Annual Technical Conference, pp. 507–522 (2016)