International Journal of Multimedia and Ubiquitous Engineering
Vol.9, No.2 (2014), pp.199-212
http://dx.doi.org/10.14257/ijmue.2014.9.2.20

BiShard Parallel Processor: A Disk-Based Processing Engine for Billion-Scale Graphs

Kamran Najeebullah, Kifayat Ullah Khan, Waqas Nawaz and Young-Koo Lee

Department of Electronics and Computer Engineering
Kyung Hee University, Korea

{kamran, kualizai, wicky786, yklee}@khu.ac.kr

Abstract

Processing very large graphs efficiently is a challenging task. Distributed graph processing systems process the billion-scale graphs efficiently but incur overheads of partitioning and distribution of the large graph over a cluster of nodes. In order to overcome these problems a disk-based engine, GraphChi was proposed recently that processes the graph in chunks on a single PC. GraphChi significantly outperformed all the representative distributed processing frameworks. Still, we observe that GraphChi incurs some serious degradation in performance due to 1) high number of non-sequential I/Os for processing every chunk of the graph; and 2) limited parallelism to process the graph. In this paper, we propose a novel engine named BiShard Parallel Processor (BSPP) to efficiently process billions-scale graphs on a single PC. We introduce a new storage structure BiShard. BiShard divides the large graph into subgraphs and maintains the in and out edges separately. This storage mechanism significantly reduces the number of non-sequential I/Os. We implement a new processing model named BiShard Parallel (BSP) on top of Bishard. BSP exploits the properties of Bishard to enable full CPU parallelism for processing the graph. Our experiments on real large graphs show that our solution significantly outperforms GraphChi.

Keywords: *Graph processing; Disk-based processing; Parallel processing; BiShard Parallel*

1. Introduction

Graph processing has been a popular research area in the last decade and a lot of research [1, 2, 3, 4] has been targeted at the most common graph processing algorithms such as shortest path and some variations of clustering and page rank. Algorithms like connected components and minimum cut also have their own vital values. In recent past, graphs gain more importance in the research community with the introduction of social networks and other very large graphs such as biological and chemical compounds. Such graphs are difficult to process because of their massive sizes. With the growing size of graph datasets, processing graphs has become more challenging.

Distributed data processing frameworks such as MapReduce [10] has been applied to graphs by many studies [11, 12], but many authors [13, 15, 16] observed that it can lead to suboptimal performance and usability issues. To solve the inherent performance problems of distributed frameworks, a range of graph specific distributed processing frameworks have been proposed [13, 14]. Pregel [13] and PowerGraph [14] are distributed graph processing frameworks based on the vertex-centric approach. This approach allows users to write an update-function without knowing any details about the distributed programming. This

ISSN: 1975-0080 IJMUE
Copyright ⓒ 2014 SERSC

function is executed for every vertex of the graph in parallel. Distributed frameworks for graph processing depict good performance with great scalability. However, they require the graph to be partitioned and distributed over the cluster of nodes. Partitioning the graph for all kinds of graph processing algorithms is a hard problem [17]. Distributed systems also require good care on part of cluster management and fault tolerance.

To solve these problems, GraphChi [18] have been proposed recently. GraphChi processes very large graphs on a single PC by using an asynchronous model, based on vertex-centric approach. GraphChi has introduced a novel processing model, Parallel Sliding Window (PSW) that processes the graph in execution intervals. Every execution interval consists of three steps 1) load a subgraph of the given large graph into memory; 2) process the subgraph and update the graph by modifying the vertices and edges; and 3) write the updated parts of the graph back to the disk. GraphChi significantly outperforms the distributed processing systems [14, 15] on per node basis.

We observe PSW inherits two serious bottlenecks. First, PSW divides the graph into several chunks, in a way that every chunk can fit into the memory. In every execution interval, PSW needs to read from all the chunks of the graph, incurring a non-sequential I/O for each of them. For a very large graph, divided into a big number of chunks, the number of non-sequential I/Os is significantly high. Second, following the vertex-centric approach, GraphChi needs to process all the vertices of a subgraph in parallel. Since GraphChi maintains only one copy of the edges, every edge can be accessed by both of its endpoint vertices. If both the endpoints of an edge are in the same interval, they cannot be processed in parallel, as they might access the common edge at the same time, leading to a race condition. In order to avoid the race conditions, GraphChi marks the common edges as critical and processes their source and destination vertices sequentially. If the graph is dense and all the edges have both endpoints in the same subgraph, all the vertices will have to be processed sequentially without any parallelism.

We propose a disk-based graph processing engine named *BiShard Parallel Processor* (BSPP) with an asynchronous model of computation to process billion-scale graphs on a single PC. We introduce a new processing model *BiShard Parallel* (BSP) based on vertex-centric approach. BSP is implemented on top of a new storage structure called *BiShard* (BS). BS divides the graph into several subgraphs. For every subgraph, it manages the in and out edges separately, which allows to load a subgraph with only two non-sequential reads. This storage structure manages two copies of every edge (one is each direction). This setting allows every vertex to have its own copy of the edges, and ensures full parallel processing of the vertices. BSP processes the graph one interval at a time; on its turn, an interval is loaded, processed and written back to disk. Our contributions are as follows:

1. A new I/O efficient storage mechanism that reduces the cost of non-sequential I/Os significantly.
2. A new processing model that exploits full CPU parallelism
3. Extensive experiments on real large graphs to show that our solution significantly outperforms state-of-the-art.

BSPP requires more disk space as compared to GraphChi, as it manages two copies of every edge. However, we believe that secondary storage is not expensive and billion-scale graphs can be stored in few gigabytes of disk space.

Rest of this paper is organized as follows. Section 2 reviews related works. Section 3 describes preliminaries. Section 4 lists the core idea of the paper in details. Section 5

 Copyright ⓒ 2014 SERSC

discusses experimental settings, results and comparison with state-of-the-art. Finally Section 6 summarizes and concludes the paper.

2. Related Work

BiShard Parallel focuses on three most related areas: I/O efficient storage techniques, asynchronous vertex-centric model and big data processing on a single machine.

Graph databases [22, 23, 24, 25] provide mechanism for efficient storage of the graph with some added facilities. InfiniteGraph [23] is designed to support distribution of data over a cluster of nodes. DEX [22] implements some basic graph processing algorithms such as shortest path and connected components. Graph databases focus on storage and querying of graphs on disk. However, they do not provide powerful processing mechanism.

Distributed graph processing frameworks like Pregel [13] and PowerGraph [14] processes the graphs by dividing and distributing them over a cluster of computer nodes. Both of them use a vertex-centric approach for efficient graph computations. Other authors [20, 15] have also observed the expressiveness of vertex-centric approach in a wide range of graph processing problems. Although we are not interested in distributed processing, as we intend to process the graph on a single machine, our point of interest in these works is vertex-centric approach that can be adopted in disk-based processing models.

Pearce *et al.* [21] proposed an efficient disk-based technique for the graph traversal based on asynchronous model of computation. Graph is stored on disk in a *Compressed Sparse Row* format. Vertex values are stored in memory, leading to higher memory requirement. Tasks are scheduled using concurrent work queues. However, their focus is limited to the graph traversal and changes to the graph are not allowed.

GraphChi [18] extended the work of Bender [26], Haveliwala [28] and Chen *et al.* [27] and proposed a mechanism that stores and process billion-scale graphs on a single consumer PC. GraphChi implements a novel technique for disk-based graph processing called Parallel Sliding Window (PSW). PSW exploits sequential I/Os and parallel computation using vertex-centric approach to efficiently process the graph. Their results show that GraphChi out-performs all the representative disk-based distributed systems [7, 9, 11, 14] on per node basis. However, experimental results also show some bottlenecks in part of parallel graph processing and number of disk reads for very large graphs.

Recently, another disk-based graph processing framework TurboGraph [19] was proposed. TurboGraph is also designed to process very large graphs on modern consumer level PC with a flash drive. It implements a novel technique called Pin and Slide. TurboGraph outperforms GraphChi by overlapping the steps of loading, processing and writing the graph to the disk. TurboGraph fully exploits the parallelism of flash disk and multi-core CPU. Their results show that their system outperforms GraphChi by order-of-magnitude. However, their solution exploits specific properties of the flash disks, which are not available on rotational disks.

3. Preliminaries

In this section we list the required preliminaries that help in understanding the problem and our proposed solution.

3.1. Disk-based processing

Here we list the constraints and challenges involved in disk based processing. In disk-based processing the size of data that need to be processed is more than the size of available memory. In practice, size of the data is order of magnitude to the size of memory. In our

Copyright ⓒ 2014 SERSC

problem we assume that; 1) a graph cannot be fully loaded into memory; and 2) any single vertex of the graph can be loaded into memory with all its edges and their associated values. Consequently, graph need to be divided and processed in chunks. However, partitioning the graph is a hard problem, as graphs along their massive size also inherently contain structural information. Graph algorithms process the graphs with respect to their structure. Without careful partitioning, disk-based graph processing algorithms incur large number of random disk I/Os.

3.2. Vertex-centric approach

Pregel [13] first introduced the vertex-centric approach and established the "think-like-a-vertex" philosophy. In vertex-centric approach user specifies an update-function that is executed for all the vertices of the graph in parallel. Vertices are allowed to change their own associated values and the associated values of their incident edges.

Many authors [20, 15, 13, 14] observed that vertex-centric approach is an efficient way to solve a wide range of graph processing problems. This approach is mostly implemented by two type of frameworks namely, synchronous and asynchronous frameworks.

Synchronous implementations are mostly based on Bulk Synchronous Parallel [29] and Message Passing Interface [30].These implementations perform computations in a number of iterations. Synchronous model incurs an expensive synchronization step after each iteration. On the other hand, asynchronous frameworks do not require any synchronization step. Changes are made to the graph directly and the subsequent iterations can access the latest updated graph. Our work is based on asynchronous implementation of vertex-centric approach.

3.3. Parallel Sliding Window

GraphChi proposed Parallel Sliding Window (PSW) for efficient processing of very large graphs with mutable edges on a single PC. PSW is based on asynchronous implementation of vertex-centric approach. PSW divides the input graph into chunks. Computation is performed in execution intervals. In every execution interval a subgraph of a very large input graph is loaded into memory. In order to load a subgraph into the memory, the engine needs to read from all the chunks of the graph. Loaded subgraph is then processed by executing update-function for all the vertices in parallel. Finally, after computation, the updated parts of the graph are written back to the disk.

4. BiShard Parallel Processor

This section describes our proposed solution BiShard Parallel Processor (BSPP). BSPP is an asynchronous disk-based framework for processing very large graphs on a single PC. First we define the notions used in our proposed system in Section 4.1. In Section 4.2 we introduce an efficient storage mechanism named BiShard. We propose a novel graph processing model called BiShard Parallel (BSP) in Section 4.3. Then in Section 4.4 we analyze the I/Os cost of the proposed engine.

4.1. Notions

Here we formally define the notions used in our proposed system.

Definition 1. Interval Given a graph G with a set of vertices V, an interval p is a subset of the set of all vertices V i.e. $p \subseteq V$

 Copyright ⓒ 2014 SERSC

Definition 2. In-Shard Given an interval p an in-shard(p) is a subset of set of all edges E. Such that every edge $e = (u_i, v_i) \in$ in-shard(p) has its destination endpoint in interval p i.e. $v_i \in p$

Definition 3. Out-Shard Given an interval p an out-shard(p) is a subset of set of all edges E. Such that every edge $e = (u_i, v_i) \in$ out-shard(p) has its source endpoint in interval p i.e. $u_i \in p$

Definition 4. Vertex Value Given a graph G with a set of vertices V, a vertex value φ_i is a user defined value associated with every vertex $v \in V$. Vertex values can be provided along the input graph and can also be computed by a graph processing algorithm.

Definition 5. Edge Value Given a graph G with a set of directed edges E, an edge value θ_i is a user defined value associated with every edge $e \in E$. Edge values can be provided along the input graph and can also be computed by a graph processing algorithm.

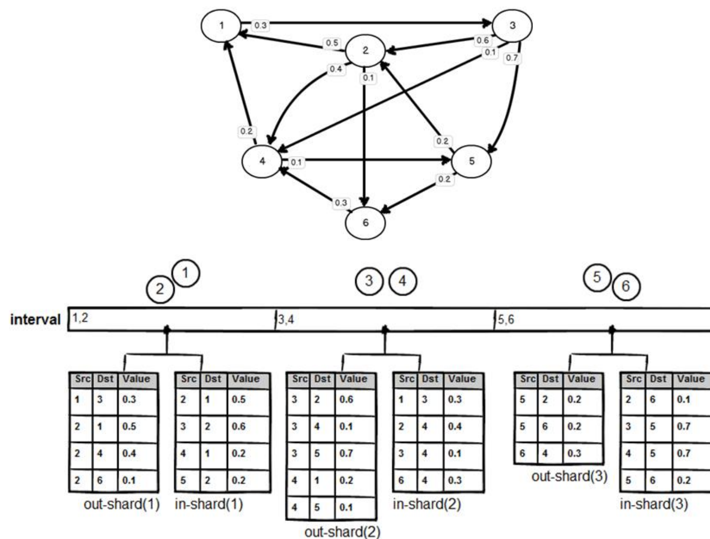

Figure 1. Graph G divided into 3 intervals. Each associated with a couple of shards. One each for in and out-edges of the interval vertices.

4.2. BiShard storage structure

We now introduce a new graph storage structure BiShard (BS). Given a weighted directed graph G with a set of vertices V and a set of edges E. A vertex value φ_i is associated with every vertex $v \in V$ and an edge value θ_i is associated with every edge $e = (u, v) \in E \; where \; u, v \in V$ i.e. $G = (V, E, \varphi, \theta)$. For a directed edge $e = (u, v)$, we refer e as in-edge of the vertex v and out-edge of the vertex u.

Given graph G, BiShard divides the graph into P intervals. Intervals are created such that number of edges across all the intervals is roughly equal. Number of intervals P is defined such that all the edges of any one interval $p \in P$ can be loaded into the memory. Every interval p is associated with a couple of shards, *in-shard(p)* and *out-shard(p)* containing all the incident edges of the interval vertices sorted by the source vertex id. Figure 1 shows an example of how a graph is divided into intervals and how their edges are stored in the corresponding in and out shards.

BiShard has two advantages over the single shard storage structure introduced by the GraphChi. One, by maintaining in and out edges separately, we make it easier to access them separately and the engine does not need to read from all the shards for every interval. Two, as we maintain two copies of every edge one in each direction, this setting allows every vertex to access its edges without any race condition.

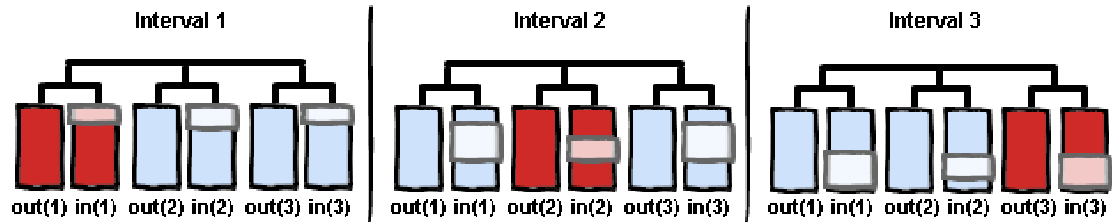

Figure 2. Visualization of full graph processing in three execution intervals. In every execution interval, the in and out-shards(in dark color) of that particular interval are loaded into memory. The blocks which are written back to all the in-shards are shown as windows on top of every in-shard

4.3. BiShard Parallel (BSP) processing model

In this section we describe a new processing model BiShard Parallel (BSP) in detail. BSP processes the graph one interval at a time. Execution of an interval consists of three steps: 1) load a subgraph of the given input graph from disk; 2) perform computation on the subgraph and modify the vertex and edge values; and 3) write the updated vertices and edges values back to the disk. Algorithm 1 lists the pseudo-code for the main execution of BSP.

4.3.1. Subgraph loading: In order to process the vertices of an interval p, first we need to read all its edges along their values from the disk. We load the in-edges from the in-shard(p) and the out-edges from the out-shard(p) and construct the interval subgraph by assigning the edges to their respective vertices. As edges are sorted by source id in both in and out-shards, the edges for each vertex are stored in consecutive chunks in both in and out shards. BSP required only 2 non-sequential disk reads to fully process an interval subgraph, irrespective of the total number of the intervals P. Algorithm 2 provides the pseudo-code for loading and constructing interval subgraph.

Algorithm 1. BiShard Parallel main execution

Algorithm 1 BiShard Parallel (BSP)

1: **for** $interval \leftarrow 1$ to P **do**
2: /* Load subgraph for interval, using Alg. 2. */
3: $intervalSubgraph \leftarrow LoadIntervalSubgraph\ (interval)$
4: **parallel for each** $vertex \in intervalSubgraph$ **do**
5: /* Execute user-defined update function, for each vertex in parallel */
8: $function_vertexUpdate(vertex)$
6: **end**
7: /* Persist updated out-edges to disk, using Alg. 3. */
8: $PersistUpdatesToDisk(intervalSubgraph)$
9: **end**

Copyright ⓒ 2014 SERSC

Algorithm 2. Loading interval subgraph

Algorithm 2 $LoadIntervalSubgraph(p)$

Input : Interval index number p
Global: $InShardBlocks$
Output: Subgraph of vertices in the interval p

1: /* Initialization */
2: $a \leftarrow interval[p].startPoint$
3: $b \leftarrow interval[p].endPoint$
4: $G \leftarrow InitializeMemoryForSubgraph\ (a, b)$
5: /* Load the in-edges */
6: $edgesIn \leftarrow in - shard[p].readFully()$
7: **foreach** $e \leftarrow edgesIn$ **do**
8: /* Note: edge values are stored as pointers. */
9: $G.vertex[edge.dest].addInEdge(e.source, \&e.val)$
10: **end**
11: /* Load the in-edges */
12: $edgesOut \leftarrow out - shard[p].readFully()$
13: /* Initialize in-shard blocks */
14: $inShardBlocks \leftarrow initializeInShardBlocks(P)$
15: **foreach** $e \leftarrow edgesOut$ **do**
16: $inShardBlocks[findTargetInShard(e.dest)].addEdgeToBlock(e)$
17: $G.vertex[e.src].addOutEdge(e.dest, \&e.val)$
18: **end**
19: **return** G

4.3.2. In-memory processing: We follow vertex-centric approach for processing the interval vertices. After the subgraph is completely loaded into memory, we run user defined update-function for all the vertices in parallel. As BSP manages two copies of every edge (one in each direction), every vertex has its own copy of the edges. This storage technique naturally eliminates race conditions, as there are no two vertices sharing a single copy of the edge. We utilize full CPU parallelism by eliminating the race conditions for accessing the edges. During processing, depending upon the computation of the update function vertices may modify their own values and the values of their incident out-edges.

For data manipulation, we follow the practice of GraphChi and keep the loaded edge values in form of blocks inside the memory, edges are referenced as pointers to the blocks. When values of the edges are modified, changes are made directly to the blocks using pointers. Algorithm 4 defines an example update-function that computes PageRank of an input graph.

Algorithm 3. Persisting updates to disk

Algorithm 3 $PersistUpdatesToDisk()$

Global: $inShardOffsets[P]$, $InShardBlocks[P]$

1: **for** $currentBlock \leftarrow 1\ to\ P$ **do**
2: /* Keep track of the offset for subsequent intervals */
3: $nextOffset \leftarrow inShardOffsets[currentBlock] + InShardBlocks[currentBlock].edges.size()$

Copyright ⓒ 2014 SERSC

4: /* Write to in-shard at given offset */
5: $writeAtOffset(InShardBlocks[currentBlock],$
$currentBlock, inShardOffset[currentBlock])$
6: /* Update offset */
7: $inShardOffset[currentBlock] \leftarrow nextOffset$
8: **end**

Algorithm 4. Example update-function

Algorithm 3 $Weight\ PageRank(vertex)$

1: **var** sum 0
2: **for** $e\ in\ vertex.inEdges()$ **do**
3: sum += $e.weight\ *\ e.weight\ *\ e.neighbor\ rank$
4: $vertex.setValue(0.15\ +\ 0.85\ *\ sum)$
5: **for** $e\ in\ vertex.outEdges()$ **do**
6: $e.neighbor\ rank\ =\ vertex.getValue()$
7: **end**

4.3.3. Persisting updates: BSP follows asynchronous model of computation. In this computation model updates to the original data are made available immediately for any subsequent processing. We follow asynchronous model of computation as it has proved to be more efficient than the synchronous model as observed by GraphChi and others [15, 35].

As every edge can be accessed by both of its endpoints, out-edge values updated while processing one interval may need to be accessed as in-edge values while processing another interval. Following asynchronous model, the updated edge values must be written back to the disk immediately in order to be available to any subsequent execution intervals.

Earlier, while loading the subgraph, we create blocks for all the in-shards inside the memory. We divide and distribute the loaded out-edges to these blocks, in the same order as they are stored in the in-shards. During on full iteration over the graph, we manage the offsets of the in-shards where the updated values need to be written. In each execution interval we commit non-sequential disk writes proportional to the number of intervals. Figure 2 shows a high level visualization of the reading and writing process in a full iteration over the graph. Algorithm 3 lists the pseudo-code that defines the process of persisting the updates to disk.

4.4. I/Os cost analysis

For the sake of meaningful comparisons we use the same I/O analysis model of Aggarwal and Vitter [31] as used by GraphChi. In this model, cost of an algorithm is the amount of data block transfers from disk to memory. If B is the size of the data block (stated in the unit of edge object), then the amount of data block transfers can be calculated by dividing total amount of data transfer by B. For the ease of computation, we assume $|E|$ is a multiple of B and number of edges in each interval $|E|/P$ are equal.

During one full pass over the graph, an edge is read exactly twice from the disk once in each direction. The numbers of disk writes are exactly half of the number of read, as we only write out-edges back to the disk. The number of reads and writes remains same irrespective of the number of inter-interval edges. At common (worst) case we add the number of seeks to the total number of block transfers. In every execution interval our solution requires exactly 2 non-sequential seeks to load edges from in and out-shards. Thus, disk seeks for one full pass

 Copyright ⓒ 2014 SERSC

over the graph as a rough cost of $\theta(2P)$. Assuming that every interval can be loaded into memory with all its vertices and edges, the I/O cost of our solution $Q_B(E)$ can be stated as

$$\frac{3|E|}{B} \leq Q_B(E) \leq \frac{3|E|}{B} + \theta(2P)$$

Note our best case I/Os cost is significantly higher than that of GraphChi. However, at best case, GraphChi assumes all the edges have both the endpoints inside the same interval. This implies that all the vertices will be processed sequentially with no parallelism.

5. Experimental Evaluation

We evaluate our engine by executing graph processing algorithms over large real graphs. We then compare our results with the state-of-the-art to show the significance of our proposed solution.

5.1. Experiments setup

Experiments were performed on a compatible server with 12 3.3GHz Intel Core i7-3960X processors, 36GB of installed main memory and a 1TB 7200rpm hard drive. We ran Microsoft Windows 7 64-bit with default settings. File system caching was disabled to get meaningful comparisons between small and large input files.

We used three real graph datasets for our experiments, LiveJournal [33], Pokec [32] and twitter-2010 [34]. Some statistics of these datasets are listed in Table 1.

Table 1. Description of datasets

Dataset	No. of vertices	No. of Edges	Description
LiveJournal	4,847,571	68,993,773	LiveJournal online social network
Pokec	1,632,803	30,622,564	Pokec online social network
twitter-2010	41,652,230	1,468,365,182	Twitter online social network

5.2. Page rank algorithm

The purpose of this experiment is to evaluate the performance of BSPP and to perform comparison with the GraphChi. Page rank algorithm is a good choice for this purpose as it scans through the entire graph and access all the in and out edges of the graph. In order to measure the effectiveness of our proposed framework we performed following experiments

1) Comparison of execution time with varying number of intervals
2) Comparison of execution time with varying number of CPU threads

5.2.1. Varying the number of intervals: We repeat the experiment with increasing number of intervals. This setting allows us to measure the relationship between the number of graph chunk and the performance of the engines. We keep the number of threads fixed to 6 for LiveJournal and Pokec datasets and 8 for Twitter 2010 dataset to eliminate the impact of increasing number of threads.

Copyright ⓒ 2014 SERSC

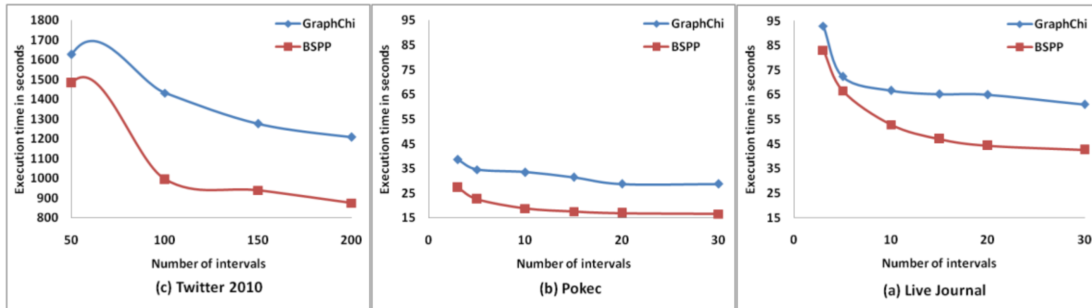

Figure 4.Graphical depiction of the performance of BSPP and GraphChi with varying number of intervals.

We notice that BSPP outperforms GraphChi at all variations of the number of intervals. Initially, when the number of intervals is small, the number of edges having both endpoints in same interval is high; this hinders GraphChi from full parallel processing of the subgraphs. BSPP handles this situation better by fully utilizing the available CPU threads. For increasing number of intervals, the performance of GraphChi shows small improvements as the number of inter-interval edges decreases but at the same time the number of random I/Os increases. On the other hand, the performance of BSPP becomes roughly consistent, proving its independence from the number of intervals. The results also show that BSPP struggles with the large size intervals and the performance margin is relatively smaller for larger intervals.

5.2.2. Varying the number of threads: We repeat the experiment with increasing number of CPU threads. This setting enables us to note how BSPP and GraphChi utilize the parallelism of available threads. We keep the number of intervals fixed to 10 for LiveJournal and Pokec datasets and to 100 for Twitter 2010 dataset to keep the number of inter-intervals edges constant.

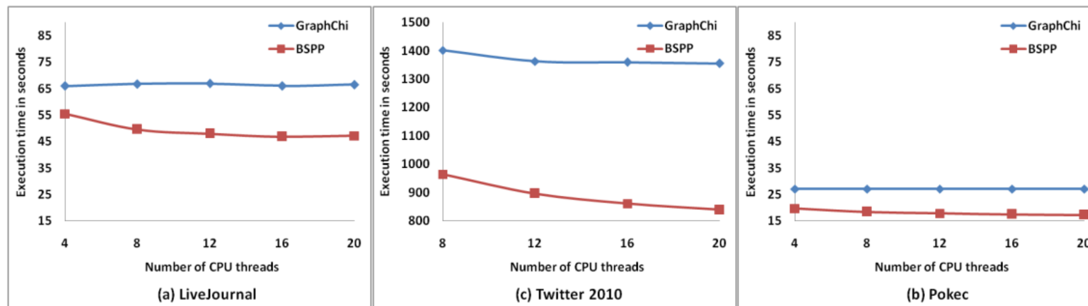

Figure 4. Graphical depiction of BSPP and GraphChi performance with varying number of CPU threads

We observer that BSPP utilize the parallelism of available threads efficiently and outperforms GraphChi by larger margins with increasing number of CPU threads. The performance of GraphChi remains roughly constant, which shows its lack of utilization of the available CPU threads.

Copyright ⓒ 2014 SERSC

6. Conclusion

Graph specific distributed processing frameworks such as Pregel are scalable and efficient, but require graph partitioning and cluster management overhead. GraphChi a disk-based single PC graph processing engine, solves the problems observed in distributed processing system, but suffers from serious performance issues. In this work, we proposed a new processing model named BiShard Parallel (BSP). We showed by theoretical analysis that our solution significantly reduced the number of non-sequential I/Os incurred in GraphChi. We also eliminated the race conditions between the vertices to access a common edge, which hindered GraphChi from full parallel processing of the vertices. We further showed by experimental evaluation and I/Os cost analysis that our solution outperforms the current state-of-the art. Our solution is not specific to any domain and can be applied to any graph processing problem that can be solved by using vertex-centric processing approach. Finally, we observe that a better performance can be achieved by overlapping the steps of an execution interval.

Acknowledgements

This research was supported by the MSIP (Ministry of Science, ICT & Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (NIPA-2013- H0301-13-4006) supervised by the NIPA(National IT Industry Promotion Agency).

References

[1] A. V. Goldberg and C. Harrelson, “Computing the shortest path: A* search meets graph theory”, In Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, (**2005**), pp. 156-165,Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
[2] S. E. Schaeffer, “Survey: Graph clustering, Computer Science Review”, vol. 1, (**2007**), pp. 27-64.
[3] S. M. Kumari and N. Geethanjali, “A Survey on Shortest Path Routing Algorithms for Public Transport Travel”, Global Journal of Computer Science and Technology, vol. 9, (**2010**), pp. 73-76.
[4] N. Duhan, A. K.Sharma and K. K.Bhatia, “Page Ranking Algorithms: A Survey, Advance Computing Conference”, IEEE International, (**2009**) March 6-7, pp. 1530-1537.
[5] K. Mehlhorn and S. Näher, “The LEDA Platform of Combinatorial and Geometric Computing”, Cambridge University Press, (**1999**).
[6] J. G. Siek, L. -Q. Lee and A. Lumsdaine, “The Boost Graph Library: User Guideand Reference Manual”, Addison Wesley, (**2002**).
[7] R. Power and J. Li, “Piccolo: building fast, distributed programs with partitioned tables”, Proceedings of the 9th USENIX conference on Operating systems design and implementation, (**2010**) October 04-06, pp. 1-14, Vancouver, BC, Canada.
[8] E. Deelman, G. Singh, M. -H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob and D. S. Katz, “Pegasus: A framework for mapping complex scientific workflows onto distributed systems”, Scientific Programming, vol. 13, no. 3, (**2005**) July, pp. 219-237.
[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, “Spark: cluster computing with working sets”, Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, (**2010**) June 22-25, pp. 10-10, Boston, MA, USA.
[10] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Proc. of6th USENIX Sym. on Operating Sys. Design and Impli., (**2004**), pp. 137-150.
[11] U. Kung, C. E. Tsourakakis and C. Faloutsos, “Pegasus: A Peta-Scale Graph Mining System - Implementation and Observations”, Proc. Intl.Conf. Data Mining, (**2009**), pp. 229-238.
[12] J. Cohen, “Graph Twiddling in a MapReduce World, Comp. in Science & Engineering, (**2009**) July/August, pp. 29-41.
[13] G. Malewicz, M. H. Austern, A. J. Bik, J. Dehnert, I. Horn, N. Leiser and G. Czajkowski, “Pregel: a system for large-scale graph processing, SIGMOD.ACM, (**2010**).
[14] J. Gonzalez, Y. Low, H. Gu, D. Bickson and C. Guestrin, “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”, OSDI, (**2012**) October, Hollywood, CA.

Copyright ⓒ 2014 SERSC

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson,C. Guestrin and J. M. Hellerstein, “Graphlab: A new parallel framework for machine learning”, UAI, (**2010**) July.
[16] R. Chen, X. Weng, B. He and M. Yang, “Large graph processing in the cloud, Proceedings of the 2010ACM SIGMOD International Conference on Management of data”, SIGMOD. ACM, (**2010**), pp.1123–1126, Indianapolis, Indiana, USA.
[17] J. Leskovec, K. Lang, A. Dasgupta and M. Mahoney, “Community structure in large networks: Natural clustersizes and the absence of large well-defined clusters”, Internet Mathematics, vol. 6, no. 1, (**2009**), pp. 29–123.
[18] A. Kyrola, G. Blelloch and C. Guestrin, “GraphChi: large-scale graph computation on just a pc”, OSDI, (**2012**), pp. 31–46.
[19] W. -S. Han, L. Sangyeon, K. Park, J. -H. Lee, M. -S. Kim, J. Kim and H. Yu, “TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single pc”, Proceedings of the 19th ACM SIGKDD Conference on Knowledge Discovery and Data mining, ACM, (**2013**).
[20] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao and E. Chen, “Kineograph: taking the pulse of a fast-changing and connected world”, Proceedings of the 7th ACM European conference on Computer Systems, EuroSys’12, ACM, (**2012**), pp. 85–98, Bern, Switzerland.
[21] R. Pearce, M. Gokhale and N. Amato, “Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory”, Super Computing, (**2010**).
[22] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M. -A. Sánchez-Martínez and J. -L. Larriba-Pey, “Dex: high-performance exploration on large graphs for information retrieval”, Proceedings of the sixteenth ACM conference on Conference on information and knowledge management, (**2007**) November 06-10, Lisbon, Portugal.
[23] INFINITEGRAPH: DISTRIBUTED GRAPH DATABASE, www.infinitegraph.com/.
[24] KOBRIX SOFTWARE, Directed hypergraph database, http://www.hypergraphdb.org/index.
[25] NEO TECHNOLOGY, Java graph database, http://neo4j.org/
[26] M. Bender, G. Brodal, R. Fagerberg, R. Jacob and E. Vicari, “Optimal sparse matrix dense vector multiplication in the i/o-model”, Theory of Computing Systems, vol. 47, no. 4, (**2010**), pp. 934–962.
[27] Y. Chen, Q. Gan and T. Suel, “I/O-efficient techniques for computing PageRank”, Proceedings of the eleventh international conference on Information and knowledge management”, ACM, (**2002**), pp. 549–557, McLean, Virginia, USA.
[28] T. Haveliwala, “Efficient computation of pagerank”, Technical Report, Stanford InfoLab, Stanford, (**1999**).
[29] L. Valiant, “A bridging model for parallel computation. Communications of the ACM”, vol. 33, no. 8, (**1990**), pp. 103–111.
[30] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra and S. Huss-Lederman, “MPI: The Complete Reference”, MIT Press, Cambridge, MA, USA, (**1995**).
[31] A. Aggarwal and J. S. Vitter, “The input/output complexity of sorting and related problems”, Communications of the ACM, vol. 31, 9, (**1988**), pp. 1116–1127.
[32] L. Takac and M. Zabovsky, “Data Analysis in Public Social Networks”, International Scientific Conference & International Workshop Present Day Trends of Innovations, (**2012**) May, Lomza, Poland.
[33] L. Backstrom, D. Huttenlocher, J. Kleinberg and X. Lan, “Group Formation in Large Social Networks: Membership, Growth, and Evolution”, KDD, (**2006**).
[34] H. Kwak, C. Lee, H. Park and S. Moon, “What is Twitter, a social network or a news media?”, Proceedings of the 19th international conference on World wide web (WWW '10), ACM, New York, NY, USA, (**2010**).
[35] D. P. Bertsekas and J. N. Tsitsiklis, “Parallel and distributed computation: numerical methods”, PrenticeHall, Inc., (**1989**).

Authors

Kamran Najeebullah

He received his B.S degree from Gomal University, Pakistan in 2008. He is Master candidate at Department of Computer Engineering, Kyung Hee University, South Korea since September 2012. His research interests include big graph processing, data mining, web mining and graph databases

 Copyright ⓒ 2014 SERSC

Kifayat Ullah Khan

He received his B.S. from Gomal University, Pakistan and M.S. from University of Greenwich, UK, in 2005 and 2007, respectively. Since September 2011, he has been working on his PhD degree at Department of Computer Engineering at Kyung Hee University, Korea. His research interests include database, data-warehousing, On-line Analytical Processing (OLAP), and graph mining.

Waqas Nawaz

He received his B.S. and M.S. from University Institute of Information Technology and National University of Computer and Emerging Sciences, Pakistan, in 2008 and 2010, respectively. Working on his PhD degree at Department of Computer Engineering at Kyung Hee University, Korea since March 2011. His research interests include data mining, graph mining, clustering and digital image processing.

Young-Koo Lee

He received his B.S., M.S. and PhD in Computer Science from Korea advanced Institute of Science and Technology, Korea in 2002. He was Postdoctoral Research Associate at Dept. of Computer Science, University of Illinois at Urbana-Champaign, Illinois, U.S.A. from Sept. 2002 to Feb. 2004. He is a professor in the Department of Computer Engineering at Kyung Hee University, Korea since 2004. His research interests include ubiquitous data management, data mining, and databases. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Copyright ⓒ 2014 SERSC

Copyright ⓒ 2014 SERSC