GSC: Greedy Shard Caching Algorithm for Improved I/O Efficiency in GraphChi

Dagang Li*†, Zehua Zheng*
*School of Electronic and Computer Engineering
Peking University, China
†Institute of Big Data Technologies
Peking University Shenzhen Graduate School, China
dgli@pkusz.edu.cn, zhzheng@pku.edu.cn

***Abstract*—Disk-based large scale graph computation on a single machine has been attracting much attention, with GraphChi as one of the most well-accepted solutions. However, we find out that the performance of GraphChi becomes I/O-constrained when memory is moderately abundant, and from some point adding more memory does not help with the performance any more. In this work, a greedy caching algorithm GSC is proposed for GraphChi to make better use of the memory. It alleviates the I/O constraint by caching and delaying the write-backs of GraphChi *shards* that have already been loaded into the memory. Experimental results show that by minimizing unnecessary I/Os, GSC can be up to 4x faster during computation than standard GraphChi under memory constraint, and achieve about 3x performance gain when sufficient memory is available.**

***Index Terms*—Graph computaion, GraphChi, GSC.**

I. Introduction and Motivation

Graph is one of the most common data structures in computer science, which can be used to represent complicated relationship between different entities intuitively and naturally. For decades, a variety of classic graph algorithms have been proposed and have conclusively proven to be effective in solving practical problems. However, things have changed greatly since the explosion of information, classic graph algorithms like PageRank and random walk are facing challenges with such a huge amount of data. According to the statistics of Statista in 1st quarter 2017 [1], Facebook had 1.94 billion monthly active users. Since each user is a vertex in the graph, it's not hard to imagine how big the amount of edges could be, with each representing "content", "follow", "like", etc. The size of ClueWeb09 dataset [2] is about 25TB, with about one billion webpages in it. It is noteworthy that the trend of data growth is so fast, for example, webpages indexed by Google were about one million in 1998, which became one trillion ten years later in 2008 [3].

Unfortunately, some excellent general-purpose big data processing platform like MapReduce [4] and Spark [5] were proven to be unsatisfactory for large scale graph computation [6], [7], so several distributed or parallel graph computation frameworks [6], [8]–[15] were proposed. "Think like a vertex" is a vertex-centric programming model designed for large scale graph computation. This model made a great progress in recent years for the reason that the users only need to define a vertex update function, then the system will execute the computation for each vertex automatically, which significantly reduce the difficulty of distributed system programming. It is also the reason why a large number of graph algorithms are based on this model [16]. Pregel [8] and GraphLab [17] are two representative distributed systems, which use a *vertex-centric* computation model dealing with graph computation. In this model, each vertex reads data from its in-edges, then executes the vertex update function and sends the results to out-edge vertices after the vertex update execution step is finished. On the other hand, several *edge-centric* model based graph computation systems like PowerGraph [11] and X-Stream [13] were also proposed. However, this model increases the difficulty of programming since it is not compatible with vertex-centric algorithms, and [3] proved that certain algorithms like community detection can't be implemented in the edge-centric model. Recently, an *I/O-centric* model Graphene [18] was proposed, which allows users to "treat the disk data as in-memory". However, vertex-centric mode is still the current mainstream.

Usually, large scale graph computation has high entry barriers, with high demand on hardware and computation management. For example, large server cluster with huge memory and fast SSD might be necessary, and issues like graph partition and node synchronization need to be well taken care of. A disk based vertex-centric single PC graph computation system named GraphChi [14] was proposed to solve this problem. With Graphchi it becomes possible for everyone to run large scale graph computation task on their personal computers. It is worth noting that the performance of GraphChi on a Mac Mini can be comparable with a Spark cluster of 50 machines or a Stanford GPS [19] (Pregel like) cluster of 30 machines [14], which proves the efficiency of its disk based algorithm and makes it still a state-of-the-art disk based system up-to-date [3].

However, through our theoretical analysis and experiments, we found that GraphChi has issues with memory utilization, because after some point the increase in usable memory will not bring performance gain any more, which means the extra memory does not help much when the system becomes I/O constrained. VENUS [3] was proposed to solve this problem. VENUS is a vertex-centric streamlined graph computation system that can also run on a single machine. It separates read-only structure from other updatable data to reduce unnecessary I/O, because read-only data like graph structure does not need to be written back. It also adopts a streamlined processing scheme which allows to "stream in" the graph data while performing computation. However, compared to GraphChi that only needs about 1% - 2% memory of the graph size to achieve optimal performance (through our experiments in Section III), VENUS caches v-shard fully in memory in order to speed up the update, thus relies heavily on large memory to work well, therefore it is not suitable for everyone's cheap machines as GraphChi does.

Although GraphChi can work correctly with very limited memory, generally its performance will improve when more memory is added. However, unless the memory is big enough to hold the complete graph to perform "in memory computing", we found that the increase in memory after some certain point will not help much in further improvement. For example, with the SNAP-LiveJournal dataset [20], 10 iterations of PageRank will take more or less the same amount of I/O and running time when the usable memory is above 110 MB, be it 120MB or 8GB there is barely any intrinsic difference. The extra memory contributes little to the overall performance when the system is struggling with the slow I/O from hard disks, therefore we want to find a better way to use the extra memory to alleviate the I/O constraint for GraphChi, for example by caching useful data and minimizing write-backs to reduce the burden on the bottleneck of I/O.

In this work, we propose Greedy Shard Caching (GSC) algorithm, a memory caching scheme for GraphChi. Our contributions are as follows.

- Identify and explain the "memory threshold" phenomenon of GraphChi by theoretical analysis and experiments. A method to determine the threshold is also provided, which is used to set the safeguard for the greedy caching proposed in this paper.
- A Greedy Shard Caching algorithm is proposed to rearrange the memory structure of GraphChi to allow caching in-memory shards to reduce I/Os. GSC introduces no action of its own but only to delay write-backs and reduce read-outs (with the help of caching), therefore it is strictly better than standard GraphChi in the amount of I/Os induced.

II. GraphChi Essentials

The performance of disk I/O is quite different between sequential and random access. For example, our 1TB TOSHIBA 7200 rpm disk performs 186.9 MB/s in SeqQ32T1 (sequential) read test, and only 0.692 MB/s in 4KQ32T1 (random) read test. The results of write are on par with the read tests. However, due to poor locality [21], single machine graph processing system needs to face the challenge of heavy random access. GraphChi performs well because it takes advantage of fast sequential I/O instead of slow random I/O by using

Shard1

src	dst	value
1	2	0.8
2	1	0.4
3	2	1.6
6	1	0.3

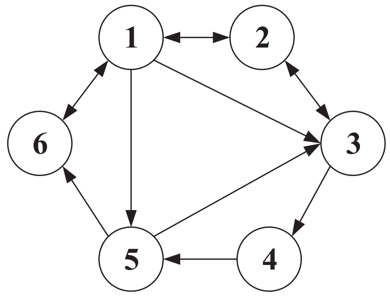
Shard2

src	dst	value
1	3	0.6
2	3	0.2
3	4	2.1
5	3	0.8

Shard3

src	dst	value
1	5	0.8
1	6	0.4
4	5	0.9
5	6	0.1

(a) Shard structure and execution of shard1



(b) Graph

Fig. 1: Example of graph sharding

a special graph cut method called *shard* and the Parallel Sliding Windows (PSW) mechanism. PSW requires only a small number of non-sequential disk I/O in the whole process.

The computation of GraphChi is divided into two stages: pre-processing stage and computing stage. In pre-processing stage, the vertices are divided into several disjoint *intervals*. Those edges with destination vertices of the same interval will be arranged into a *shard*. Since an interval could have more than one vertex, edges in each shard are sorted in the order of their source vertices, as show in Fig. 1(a). The pre-processing is kept unchanged in this paper. In computing stage, each shard is calculated one at a time. Fig. 1 (a) also shows the calculation of shard1, in which shadowed edges need to be loaded into memory, including the complete shard1 and outgoing-edges of shard1-vertices from shard2 and shard3, which are at the top of those shards. These shadowed edges will not be needed anymore after the calculation on shard1, and their updated values will be written back to the disk. With the mechanism of PSW, we could find that the next needed edges for calculating shard2 are actually the ones just beneath these shadowed ones, which makes the shadow "window" slide down in each of the shards and make the read-outs mostly sequential.

III. Memory Threshold Analysis

After the calculation on shard1, shadowed edges in Fig.1 (a) will be written back on disk with the updated values before start calculating shard2. However, some of the write-backs can be redundant. For example in Fig.1 (a), after writing back (1,3) and (2,3), they will be loaded again as part of shard2 for the calculation on shard2. The save-and-load of these two edges is obviously unnecessary and only causes extra burden on the I/O bottleneck and slows the system down. We believe

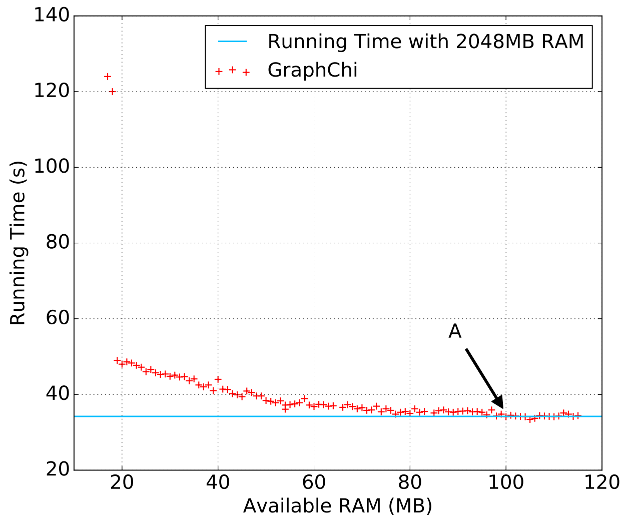


Fig. 2: Memory threshold on SNAP-LiveJournal

this behavior has strong links with the GraphChi's "memory threshold" phenomenon when the system is I/O constrained, and this observation inspires us with the idea to cache them in memory to eliminate these redundant I/Os.

We did extensive experiments on GraphChi with different available memory to explore the "memory threshold" phenomenon. The RAM block method from [7] and [22] is used to control precisely the available memory for GraphChi. The hardware configuration is as follows: Intel Core Haswell i5-4590 @3.30GHz CPU, 2*8GB DDR3 1600MHz RAM and TOSHIBA 7200rpm disk. Experiments were carried out on Ubuntu 14.04 LTS with the dataset of SNAP-LiveJournal social network [20], which has 4.8 million vertices and 69 million edges. The size of the dataset is 1.1GB. The computation task is 10 iterations of PageRank.

The available RAM in the experiments ranges from 17MB to 115MB with step of 1MB. The running time is the overall duration including both pre-processing stage and computing stage. Also, a more than sufficient RAM of 2048MB (larger than the dataset size of 1.1GB) was also tested as the reference. Results averaged over multiple runs are showed in Fig.2. We can see that when the available RAM is extremely insufficient, for example at 17MB, the running time can be very high; after the available RAM reaches 20MB, a boost in performance can be found, and the improvement continues and gradually approaches the line of 2048MB. Actually when the available RAM reaches about 100MB, the performance is almost as good as that of 2048MB of RAM: this can be regarded as *the* Memory Threshold (marked as point A in Fig. 2), after which more memory brings no performance improvement any more.

GraphChi actually has its own mechanism to adapt to the available memory: the size of shards will change and when more memory is available it will be bigger to make use of the extra memory. However if we look closer, we will find that the size of shards has little effect on total I/O, because each edge will always be loaded twice during a computation task, once as part of the main shard, the other time as the out-edge of another main shard. Experimental results verify the analysis above: Table I shows that the difference in total I/O is very small as long as sufficient memory is available (around and above the memory threshold). Extremely inefficient memory, on the other hand, leads to about 30% rise in I/O because of frequent swapping. Larger shards for larger memory can still help in I/O because they are even more sequential, but the benefit is somehow marginal.

TABLE I: I/O Amount Experiment

RAM (MB)	18	60	100	1024	2048
I/O (GB)	34.5	29.5	27.3	26.4	25.8

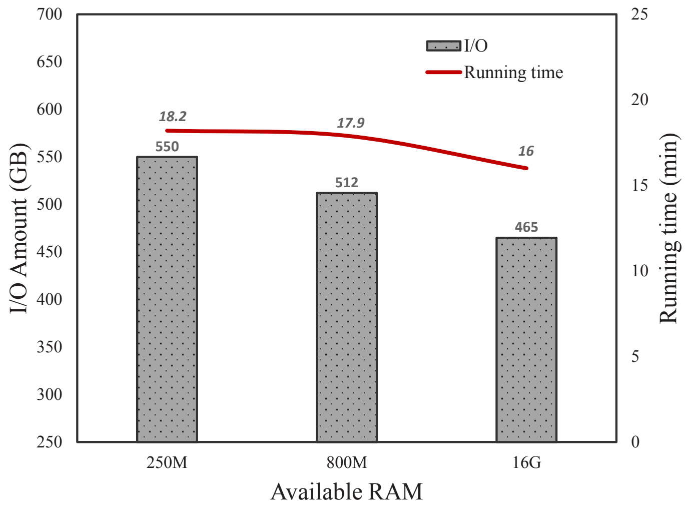


Fig. 3: Running time and I/O amount on SNAP-twitter

The Linux system also has its internal transparent page cache mechanism that is designed to save I/O for frequently visited file contents, but from the experiments we can see that it doesn't help much with the memory threshold phenomenon because its caching and flushing strategies are not optimized for the file accessing pattern of GraphChi.

We did the same experiments on a much bigger dataset: SNAP-Twitter follower network [20], which consists of 41.6 million vertices and 1468 million edges. The size of the dataset is 26.1GB, way larger than the physical memory we have, but similar trend was observed, as shown in Fig. 3. The performance at 250MB and 800MB are roughly equal in running time and extra memory of 16GB does not help too much, the trend in total I/O is also in line with expectations. From these experiments on two different datasets, we can see that the running time of GraphChi is very much determined by I/O, and the way GraphChi utilizes the extra memory (by using bigger shards) does not really help with the total I/O, therefore does not benefit the overall running time, thus the "memory threshold" phenomenon.

Here in this paper we want to use the extra memory more wisely to reduce the redundant I/O mentioned at the beginning of this section, which in turn will reduce the total running time.

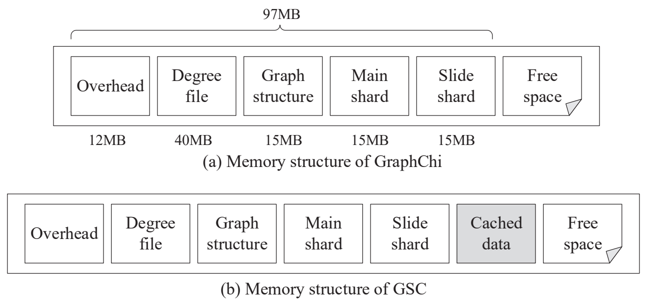


Fig. 4: Memory sturture

IV. Algorithm Design

Before we design the caching algorithm, we should know how much memory actually can be used for caching without affecting the correct execution of GraphChi.

The memory structure of GraphChi is shown in Fig.4 (a), which consists of five parts. The values below the boxes are the size of each part as measured in the experiment on SNAP-LiveJournal dataset with 100MB available RAM. The "Overhead" part is the baseline memory needed to run a task. In the experiments GraphChi can work with 13MB memory, but fails with 12MB, so we set overhead as 12MB. The "Degree file" contains in- and out- degree for each vertex, and its size does not change with the size of available memory. The "Graph structure", originally called *adjacency shard* in GraphChi, stores the edge array for each vertex of a shard. It does not store any edge value. The values are stored in shards, and each shard has a graph structure file of its own. A shard stores its edges in block files, and blocks are the basic I/O units of size 1MB. During a computation task, the currently calculated one is the "Main shard" which is fully loaded in memory. For example Shard1 in Fig.1 is the main shard at the moment. Analogously, the shadows in shard2 and shard3 constitute the "Slide shard". Because GraphChi will use larger shards with larger available memory, the latter three parts will also become larger accordingly.

The calculation of GraphChi needs all these five parts in memory, so we need to reserve enough memory for them. Fortunately their size can be easily obtained at the end of pre-processing, so we can use whatever left to do the caching. For example, with 100MB RAM as shown in Fig. 4 (a), the total size of the five parts is 97MB which does not have much room left. With 800MB RAM available the size of each shard is 65MB, and each graph structure becomes 70MB. Now the total size we need to reserve becomes 12+40+70+65+65=252MB, which leaves plenty space for us to use. In GSC, we add a new part called "cached data" to do the caching, and the new memory structure of GSC is shown in Fig.4(b).

A. Data Cache Table

The "Cache data" part is the primary data structure in GSC, which consists of an index table and a priority queue. Since shards in GraphChi are actually sets of 1MB blocks, we also choose blocks as the basic caching unit. A block can be uniquely identified by the shard number and block number, for example, shard(2)(3) means the third block in shard2. We chose STL map as the index table, which stores pointers to blocks cached in memory. If a block to be read is found in the table, we have a hit and the block is referenced directly from the cache and the read disk I/O is saved. The priority queue is used for deciding which blocks to be released from memory when there's not enough space in the cache. Traditionally, after every shard calculation Graphchi writes the modified blocks to disk and discards the rest, but they may still be useful for subsequent computations. In GSC we cache all of them, but treat them differently when cache is near full: unmodified blocks can just be discarded to recover memory, so they are preferred and arranged to the head of the priority queue; modified blocks need to be write back to disk so they will only be release when no unmodified blocks are left in cache. These blocks are arranged by the order they get cached (FIFO Scheduling), so blocks with the longest cache time get written back first. In other words, effectively GSC delays the write-back of modified blocks until necessary to reduce the write disk I/O.

B. Upper Limit on Cached Data Area

We can't use all the free memory for caching data: besides the five parts mentioned in the previous subsection, we still need to put aside enough free memory for the blocks required for the computation of the next shard but not already in the cache, or else the execution of GraphChi will be affected.

For example as shown in Fig.4(a), the data considered to be cached in GSC are the main shard and the slide shards. The overhead and degree file are static throughout the execution of GraphChi, and the graph structure is only required by the current main shard, so they don't need to be cached. In this example reserved memory size is 97MB, so there is not much free space left and GSC will not be activated. When the usable memory gets larger, for example at 800MB, then we only need to reserve 252MB and the rest can be used for GSC. However, we can't use up all the free space because there might be blocks not cached but required for the computation of next shard. In the extreme case, they can be all the blocks of the next main shard and the corresponding slide shard. Because GraphChi always partitions edges into shards of more or less the same size, and the shadows in the slide shard are also of a comparable total size, if we leave a memory space of twice the size of an average shard (enough to handle the extreme case), we are very much on the safe side with a comfortable margin because it is not very likely that all the required blocks are not in the cache from a previous computation.

C. Caching Data Greedily

Original in GraphChi, after the round of computation on shard-i, update in both shard-i and the other slide shards will be written back to disk. In GSC, these write-back operations are intercepted and if the free space is greater than $2 \times A$, no write-back will be executed and all of them are cached. If the free space is less than $2 \times A$, cached blocks will be released one by one until the free space is at least $2 \times A$: cached unmodified

```
Algorithm: Greedy Shard Caching (GSC)
1:  if Shard i finished then
2:      while $F > 2 * A$ do
3:          $C \leftarrow$ C(i)
4:          /* Logical operation, done by
5:          modifying Data Cache Table */
6:          if All C(i) have cached and $F \geqslant 2 * A$ then
7:  return
8:      /* Now F maybe deficiency for the
9:      execution of Shard i+1 */
10:     while $F < 2 * A$ do
11:         if there are unmodified blocks then
12:             Free(UnmodifiedBlock)
13:         else
14:             ModifiedBlock.ToDisk()
15:     /* Now,the free space is more than 2A */
16: return
17: Shard i+1 begin
```

Fig. 5: Pseudocode of GSC algorithm

blocks will be freed first, then modified blocks at the head of priority queue will be written back and freed. For the following rounds of computation, the cache table will be checked first before a disk read, and with a cache hit, the cached data will be used instead. The pseudocode of GSC algorithm is shown in Fig.6. Since GSC only caches data blocks already in memory and delays write-backs originally issued by GraphChi, no extra read or write I/O will be introduced. In the extreme case that there is no cache hit during the computation task, the I/O is exactly the same with GraphChi.

V. Experimental Evaluation

The hardware is exactly the same as used in Section III. The computation task is 10 iterations of PageRank. Fig. 6 shows the performance of PageRank on SNAP-LiveJournal dataset, with the running time as the sum of the pre-processing and computing stages. Since the pre-processing stage is the same for GraphChi with and without GSC, the time it takes is marked as black triangles at the bottom of the graph. The case with sufficient RAM of 2048MB was also test as comparison. From the figure we can see that GSC is more than 4x faster than GraphChi in the computing stage (about 2x faster overall) when the memory is very insufficient, which means GSC also works very well at very strict memory limitations; when the available RAM reaches 120MB and above, GSC will have a steady improvement of 300% plus in the computing stage (50% improvement overall) over the standard GraphChi.

The performance of GSC was also evaluated with several other typical graph computations: random walks (RW), weakly connected component (WCC) and triangle count (TC). GSC achieved about 60% to 300% improvement in the computing stage over standard GraphiChi, especially in TC, as shown in Fig. 7. The available RAM are set to 100MB, 500MB and 2048MB to represent different sufficiency of available memory.

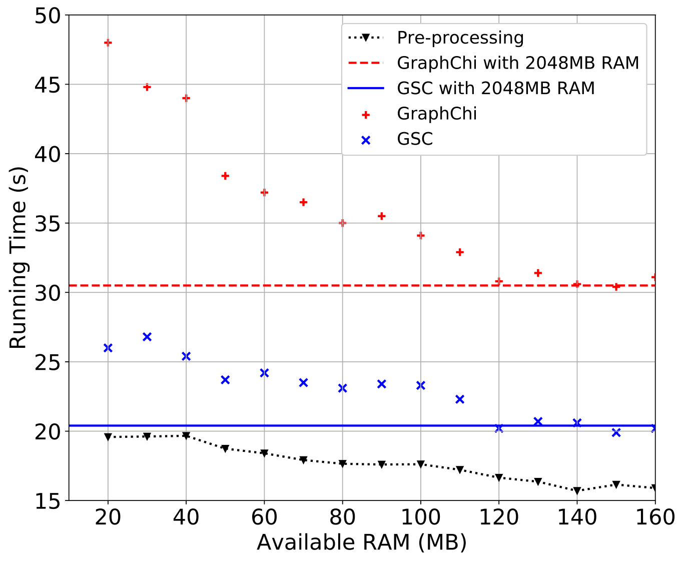


Fig. 6: PageRank on SNAP-LiveJournal

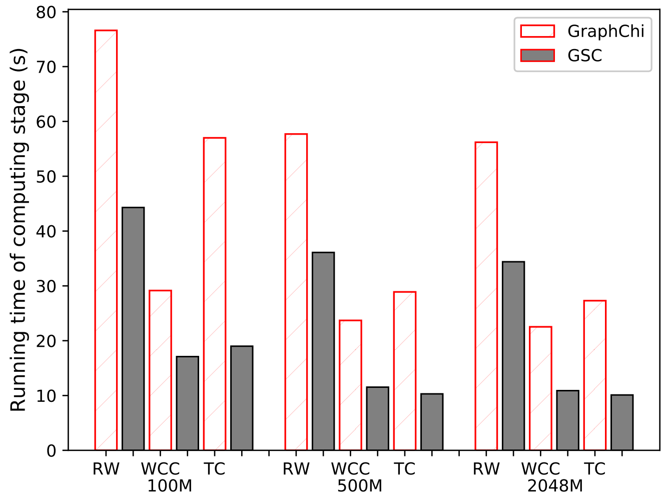


Fig. 7: Several algorithms on SNAP-LiveJournal

GSC is proposed to reduce I/O by caching the right number of blocks in memory in case of later use, so the overall running time of GraphChi can be improved. A good metric for a caching mechanism is to see how many cache hit is achieved during its execution. However, in the experiments we found that it is not only our GSC who is trading between memory and I/O, there are other mechanism such as the Linux swap mechanism in the background that is doing a similar job, but in the opposite direction. For example, when the available memory is insufficient, Linux swap mechanism will move some pages in memory to disk in the background which generates disk I/O. If the caching mechanism caches too much, and some cached blocks are swapped to disk, then a false cache hit may happen if a swapped block is accessed. GraphChi will be affected by all the I/O generated during its execution, no matter caused by GSC or swapping, so we want to see how much I/O GSC actually reduces when interacting with the Linux embedded swap mechanism.

We use the number of page faults as the measurement,

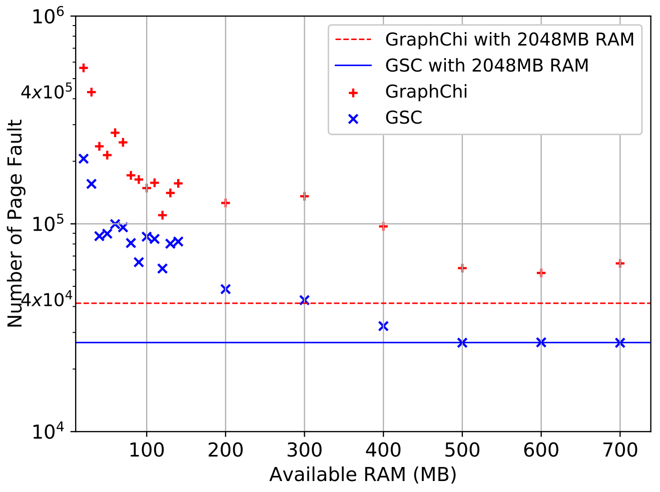


Fig. 8: Page fault experiment

since I/O generated by both swap and the mmap used by GraphChi to read file from disk can be detected by page faults. The computation task is 10 iteration of PageRank on SNAP-LiveJournal. Fig. 8 shows the results, which include both pre-processing and computing stage. The number of page faults increases vastly in order of magnitude when usable memory gets reduced. When the memory is insufficient, like 20MB, the number of page faults is 2.7x more in GraphChi than with GSC; when the memory is sufficient, the number of page faults is still at least 1.7x more with standard GraphChi. When the available memory reaches 200MB, the number of page faults in GSC is already as good as the best performance the original GraphChi can achieve with memory of 2048MB. Here the improvement in total I/O is higher than the improvement measured in total running time as shown in Fig. 6, because pre-processing takes a large portion of running time in which GSC does not cover.

Acknowledgment

This work was supported by Shenzhen Research Program JCYJ20150626110611869, Guangdong Science and Technology Project 2014B010117007 and the Shenzhen Municipal Development and Reform Commission (Disciplinary Development Program for Data Science and Intelligent Computing).

VI. Conclusions

We found a “memory threshold” phenomenon in GraphChi, which indicated that GraphChi does not make good use of memory when it is I/O constraint. A mechanism called Greed Shard Caching (GSC) was then proposed specifically for GraphChi to use the extra memory for caching to reduce I/O and improve the overall performance. GSC only uses the proper amount of memory for caching and minimizes I/O during cache replacement. Experiments on real world large scale graph datasets show that GSC can be up to 4 times faster than original GraphChi in the computing stage under very limited memory, and achieve about 3 times performance improvement in the general conditions.

References

[1] “Facebook monthly active users,” https://www.statista.com.
[2] “Clueweb09,” http://www.lemurproject.org/clueweb09.php.
[3] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. Lui, and C. He, “Venus: Vertex-centric streamlined graph computation on a single pc,” in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 1131–1142.
[4] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets.” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
[6] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, “Nxgraph: an efficient graph processing system on a single machine,” in *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 2016, pp. 409–420.
[7] S. Cheng, G. Zhang, J. Shu, Q. Hu, and W. Zheng, “Fastbfs: Fast breadth-first graph search on a single server,” in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 303–312.
[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.
[9] C. Avery, “Giraph: Large-scale graph processing infrastructure on hadoop,” *Proceedings of the Hadoop Summit. Santa Clara*, vol. 11, no. 3, pp. 5–9, 2011.
[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed graphlab: a framework for machine learning and data mining in the cloud,” *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.
[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs.” in *OSDI*, vol. 12, no. 1, 2012, p. 2.
[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “Graphx: Graph processing in a distributed dataflow framework.” in *OSDI*, vol. 14, 2014, pp. 599–613.
[13] A. Roy, I. Mihailovic, and W. Zwaenepoel, “X-stream: Edge-centric graph processing using streaming partitions,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 472–488.
[14] A. Kyrola, G. E. Blelloch, and C. Guestrin, “Graphchi: Large-scale graph computation on just a pc.” USENIX, 2012.
[15] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, “Mosaic: Processing a trillion-edge graph on a single machine,” in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 527–543.
[16] R. R. McCune, T. Weninger, and G. Madey, “Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 25, 2015.
[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, and C. Guestrin, “Graphlab: A distributed framework for machine learning in the cloud,” *arXiv preprint arXiv:1107.0922*, 2011.
[18] H. Liu and H. H. Huang, “Graphene: Fine-grained io management for graph computing.” in *FAST*, 2017, pp. 285–300.
[19] S. Salihoglu and J. Widom, “Gps: A graph processing system,” in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 22.
[20] “Snap dataset,” https://snap.stanford.edu.
[21] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, “Challenges in parallel graph processing,” *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.
[22] L. Bindschaedler and R. Amitabha, “Benchmarking x-stream and graphchi,” *LABOS, EPFL, Tech. Rep.*, 2015.