Design of a NVRAM Specialized Degree Aware Dynamic Graph Data Structure *

Keita Iwabuchi
Tokyo Institute of Technology
iwabuchi.k.ab@m.titech.ac.jp

Roger Pearce
Lawrence Livermore National Laboratory
rpearce@llnl.gov

Brian Van Essen
Lawrence Livermore National Laboratory
vanessen1@llnl.gov

Maya Gokhale
Lawrence Livermore National Laboratory
maya@llnl.gov

Satoshi Matsuoka
Tokyo Institute of Technology
matsu@is.titech.ac.jp

1. INTRODUCTION

Large-scale graph processing, where the graph is too large to fit in main-memory, is often required in diverse application fields. Recently, Non-Volatile Random Access Memories (NVRAM) devices, e.g., NAND Flash, have enabled the possibility to extend main-memory capacity without extremely high cost and power consumption. Our previous work demonstrated that NVRAM can be a powerful storage media for graph applications [4],[6]. However, constructing large graphs is expensive especially for NVRAM devices, due to sparse random memory accesses.

In many graph applications, the structure of the graph changes dynamically over time and may require online analysis. We describe a NVRAM specialized degree aware dynamic graph data structure using open addressing compact hash tables in order to minimize the number of page misses. We demonstrate that our dynamic graph structure can scale near-linearly on an out-of-core dynamic edge insertion workload.

2. DYNAMIC GRAPH DATA STRUCTURE

The key challenges of designing a dynamic graph data structure are: 1) supporting out-of-core access in order to process large-scale graphs; 2) high performance insertion and deletion of vertices and edges; 3) providing static data structure like performance on read-only tasks, such as traversing a static graph. To tackle the above challenges, we designed a NVRAM specialized degree-aware dynamic graph data structure using compact open addressing hash tables. We envision this approach as the underling graph storage for a distributed graph database.

*This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344(LLNL-ABS-676161).

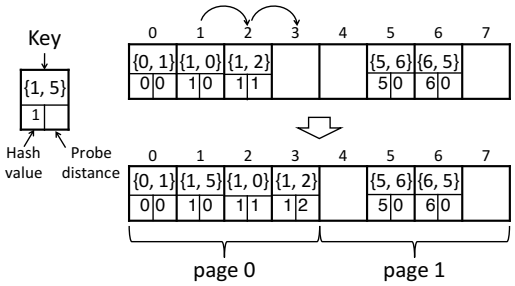

Figure 1: Edge insertion into Robin Hood Hashing table

2.1 Robin Hood Hashing

In order to minimize the number of accesses to NVRAM, resulting in page misses, we choose Robin Hood Hashing [2] because of its locality properties [5]. Robin Hood Hashing is an open address hashing that is designed to maintain a small average probe distance, the distance between ideal (initial) position and current position of a key.

An illustration of edge insertion using Robin Hood Hashing is shown in Figure 1. For example, {0, 1} means vertex 0 is connected to vertex 1. First, compute a hash value (initial position) of the new edge {1,5} using a hash function. Second, edge {1,0} is moved to the next position since the probe distance of {1,0} is equal or smaller than the one of edge {1,5}. Next, edge {1,2} is moved to the next position and edge {1,0} is inserted in the position (index 2) because probe distance of edge {1,2} is smaller than the one of edge {1,0}. Ideally, the target element is located in a same page, even if the element is moved to an another position because of hash conflict.

2.2 NVRAM Specialized Degree Aware Dynamic Graph Data Structure

An illustration of an insertion algorithm of our dynamic graph data structure is shown in Figure 2. Depending on the degree of a source vertex, edges are stored in two types of table: low-degree table and middle-high-degree table. The low-degree table stores edges in a single compact table as described in Figure 1. The middle-high-degree table stores

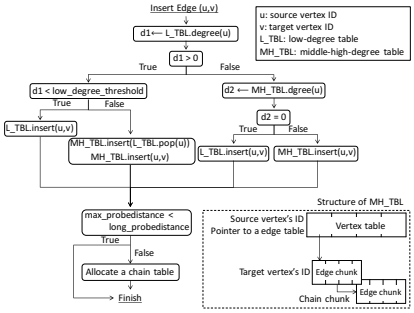

Figure 2: Insertion Algorithm of NVRAM Specialized Degree Aware Dynamic Graph Data Structure

vertex information, e.g., source vertex's ID and vertex meta data if needed, in a vertex-table. Adjacency edges are stored into edge-chunks (also hash tables) and the vertex table holds a pointer to the edge-chunks.

3. EXPERIMENTS

3.1 Experimental Setup

We used the Catalyst cluster at LLNL. Catalyst has Intel(R) Xeon(R) E5-2695v2 processors and is equipped with 800 GB of node-local PCIe NVRAM.

We implemented our degree-aware dynamic graph data structure in C++, and used the *Boost.Interprocess* library to allocate our structures in a memory mapped region. We used DI-MMAP[3], a custom memory-map runtime, as an interface to NVRAM and limited the DRAM resident portion of graph DB (page buffer) to 4 GB.

We also implemented a baseline model, for experimental comparison. The model has a vertex table which holds source vertices and a pointer pointing at an edge table using the *Boost unordered_map* container; edge tables which hold adjacent-edges using the *Boost vector* container.

Dataset: we used a large WebGraph[1], the largest open source graph to our knowledge, that has 120 billion of edges; each vertex represents a web page and edges represent hyperlinks.

Edge Reordering: we setup an experimental evaluation by reordering the edge stream into a sorted order. We also reorder the edges in a BFS order to simulate the behavior of a web crawler. We conduct a BFS from a randomly selected vertex and list all edges when its source vertex is visited. In the WebGraph2012 dataset, approximately 70 % of edges are visited in a single BFS from an chosen vertex.

Edge Partitioning: we assume that the our graph data structure uses completely separated memory spaces among the MPI processes to store a distributed graph. Thus, after the reordering, the edges are divided into 1D and 2D partitions (the number of partitions is 64). Note, since we only use a single process in this experiment, only the edges assigned to a partition are used.

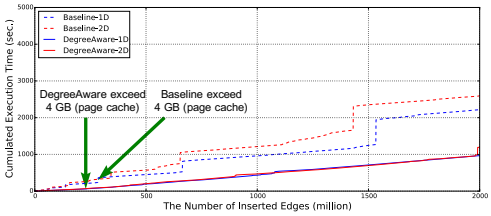

Figure 3: Insert sorted edges with 1D and 2D partitioning

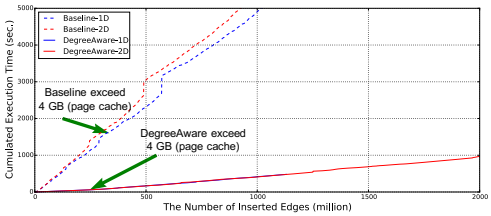

Figure 4: Insert BFS ordered edges with 1D and 2D partitioning

3.2 Results

Figure 3 shows insertion performance on sorted edges and Figure 4 shows insert performance on BFS ordered edges. As the graph size increases, the execution time of the baseline model increases rapidly in the both figures. On the other hand, our proposed data structure can insert edges with near-linear scaling, up to 2 billion edges, due to the combination of locality aware Robin Hood Hashing and a compact representation of low-degree vertices contribute to this scaling result.

4. REFERENCES

[1] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Comput. Netw.*, 33(1-6):309–320, June 2000.

[2] P. Celis. *Robin Hood Hashing*. PhD thesis, Waterloo, Ont., Canada, Canada, 1986.

[3] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale. Di-mmap: A high performance memory-map runtime for data-intensive applications. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 731–735, Washington, DC, USA, 2012. IEEE Computer Society.

[4] K. Iwabuchi, H. Sato, Y. Yasui, K. Fujisawa, and S. Matsuoka. Nvm-based hybrid bfs with memory efficient data structure. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 529–538, Oct 2014.

[5] M. Mitzenmacher. A new approach to analyzing robin hood hashing. *CoRR*, abs/1401.7616, 2014.

[6] R. Pearce, M. Gokhale, and N. Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 825–836, May 2013.