Large-Scale BSP Graph Processing in Distributed Non-Volatile Memory

Takumi Nito, Yoshiko Nagasaka, Hiroshi Uchigaito
Hitachi, Ltd., Reserch & Development Group
292, Yoshida-cho, Totsuka-ku, Yokohama-shi, Kanagawa-ken, Japan
{takumi.nito.sh, yoshiko.nagasaka.vc, hiroshi.uchigaito.nx}@hitachi.com

ABSTRACT
Processing large graphs is becoming increasingly important for many domains. Large-scale graph processing requires a large-scale cluster system, which is very expensive. Thus, for high-performance large-scale graph processing in small clusters, we have developed bulk synchronous parallel graph processing in distributed non-volatile memory that has lower bit cost, lower power consumption, and larger capacity than DRAM. When non-volatile memory is used, accessing non-volatile memory is a performance bottleneck because accesses to non-volatile memory are fine-grained random accesses and non-volatile memory has much larger latency than DRAM. Thus, we propose non-volatile memory group access method and the implementation for using non-volatile memory efficiently. Proposed method and implementation improve the access performance to non-volatile memory by changing fine-grained random accesses to random accesses the same size as a non-volatile memory page and hiding non-volatile memory latency with pipelining. An evaluation indicated that the proposed graph processing can hide the latency of non-volatile memory and has the proportional performance to non-volatile memory bandwidth. When non-volatile memory read/write mixture bandwidth is 4.2 GB/sec, the performance of proposed graph processing and the performance storing all data in main memory have the same order of magnitude (46%). In addition, the proposed graph processing had scalable performance for any number of nodes. The proposed method and implementation can process 125 times bigger graph than a DRAM-only system.

Keywords
big data; non-volatile memory; parallel algorithms; distributed computing.

1. INTRODUCTION
Big Data processing has become a powerful tool for analyzing very large real-world problems. Several programming models and software frameworks have been proposed to handle big data on distributed computer systems such as Apache Hadoop [1] and Apache Spark[2]. In addition, large-scale graph processing is becoming a useful way to handle the complex relationship between big data and data elements. Graphs are used in a wide range of fields including computer science, biology, chemistry, homeland security, and the social sciences. These graphs may represent relationships between items such as individuals, proteins, or chemical compounds. There are some software frameworks, benchmarks, libraries, and services for the graphs. For example, several software frameworks that handle large-scale graphs are Google Pregel [3], Apache Giraph [4], GraphLab [5], and GraphX[6].

If all of the large-scale data are stored in dynamic random access memory (DRAM), large-scale graph processing requires a large-scale cluster system, which has high costs and power consumption and has difficulty making efficient programs that have high parallelization. Non-volatile memory such as NAND flash memory uses less power and has larger capacity than DRAM. Additionally, the bulk synchronous parallel (BSP) model used in Pregel is a computational model well suited to distributed systems. Therefore, we examined the BSP graph processing that stores large-scale data in non-volatile memory and uses non-volatile memory as expansion of main memory in a small-scale cluster system.

The outline of the paper is as follows. First, we explain the problem of BSP graph processing with non-volatile memory. Next, we propose a non-volatile memory group access method and its implementation to solve the problem. Finally, we evaluate the proposed method and summarize our work in the conclusion.

2. BSP GRAPH PROCESSING
In this section, we outline BSP graph processing and describe the problems that occur when it is executed in non-volatile memory.

2.1 Using Distributed Non-volatile Memory
The diagram of BSP graph processing is shown in Figure 1. During a superstep each vertex calculation is executed repeatedly. There is synchronization between supersteps. The vertex calculation specifies behavior at a single vertex V and a single superstep S. It reads messages sent to V in superstep S – 1, modifies the state of V and its outgoing edges, and sends messages to other vertices that will be received at superstep S + 1. Messages are typically sent along outgoing edges.

The messages exchanged between vertices and the graph data consisting of vertices and edges that mean the structure of the graph become large in proportion to the number of vertices. For large-scale BSP graph processing in small clusters with non-volatile memory, both datasets are stored in non-volatile memory. Graph processing involves repeatedly executing the vertices

GRADES'15, May 31 - June 04 2015, Melbourne, VIC, Australia

DOI: http://dx.doi.org/10.1145/2764947.2764949

calculations on the basis of messages and the graph data of a vertex that are read from non-volatile memory before the vertex calculation. After executing the function, updated vertices, edges, and received messages are written to non-volatile memory.

In this paper, the target of the proposed method is a static graph, only the vertices and messages are changed in each superstep, and edges are not changed in the graph processing.

2.2 Random Access Problem

Graph data consisting of vertices and edges in the non-volatile memory can be accessed in address order by storing the data in calculation order of the functions of the vertices. However, the messages could not be accessed in a similar way.

The messages are sent to the vertices connected with edges of the vertex, and these messages are stored in non-volatile memory on the server node receiving message. Messages that are sent to a vertex are read from non-volatile memory to main memory before calculating the vertex function in the next superstep. Random access to non-volatile memory occurs because the order of reading messages is different from the order of storing messages in the non-volatile memory. Figure 2 shows an example of the access pattern to the messages. The calculation order of the functions is A, B, C, …, F. Message 1 says the source vertex is A and the destination vertex is B. When vertex A is calculated, messages 1, 2, 3, and 4 are sent from vertex A to B, C, D, and F. Then, the messages are sent in the order of 1, 2, …, 16. The messages are received in the order they are sent and stored in non-volatile memory sequentially. For example, messages 5, 8, 10, and 14 are necessary to calculate the function of vertex A in the next superstep, and random access occurs while these messages are read from non-volatile memory.

In non-volatile memory, a fine-grained random access that is smaller than a page of non-volatile memory becomes slow. The page size of the non-volatile memory is generally around 8–16 Kbytes. A message is smaller than a page in general graph processing. For example, in PageRank [7] processing, a message is a PageRank value of a vertex, so its size will be several bytes. When the page is 8 Kbytes and the random access is 8 bytes, the random access is 1024 times slower than the sequential access. The random accesses substantially reduce the performance of BSP graph processing in non-volatile memory.

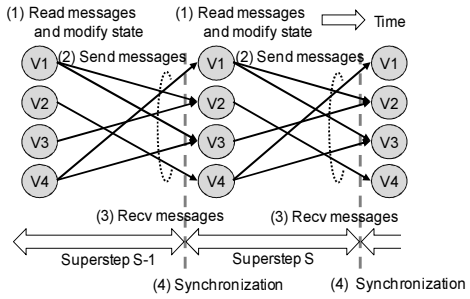

Figure 1. Diagram of bulk synchronous parallel graph processing.

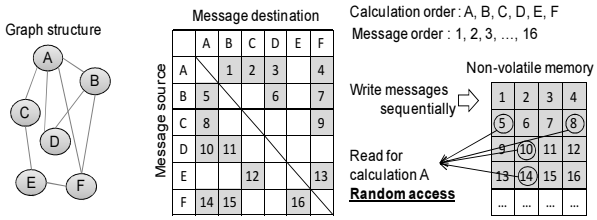

Figure 2. Example of graph structure, the order of messages between vertices, and the location of messages in non-volatile memory.

3. GROUP ACCESS METHOD

We proposed a group access method that changes fine-grained random accesses to random accesses the same size as a non-volatile memory page by devising a data location and access order of non-volatile memory. It improves the performance of accessing messages in non-volatile memory.

3.1 Group Writing

The order of storing messages is different from the order of reading them in BSP graph processing with non-volatile memory. This method groups vertices into multiple vertex groups and stores only the messages sent to vertices belonging to the same vertex group on the same page of non-volatile memory. The actual method for storing messages to non-volatile memory is depicted in Figure 3. First, the method prepares buffers equal in number to vertex groups that are the same size as a non-volatile memory page; each buffer corresponds to each vertex group. When a message to a vertex is received, the method examines to which vertex group the vertex belongs and writes a message to the buffer corresponding to that vertex group. If that buffer is full, the method writes the data of that buffer to the non-volatile memory and clears the buffer. In this way, the same page of the non-volatile memory stores the messages to vertices belonging to the same vertex group, and all of the non-volatile memory writing accesses are the same size as a non-volatile memory page (buffer size).

3.2 Group Reading and Calculation

All functions of vertices included in a vertex group are calculated at the same time; all messages sent to the vertices included in the vertex group are read from non-volatile memory to main memory. Then all functions of vertices included in the vertex group are calculated while maintaining the messages of the group in the main memory. After all calculations of vertices belonging to the group are completed, the messages of that group are discarded and the next group message reading and calculation is started. It is not necessary to read the same non-volatile memory several times in a superstep because all vertices included in a vertex group are calculated simultaneously. In addition, all non-volatile memory reading accesses are the same size as a non-volatile memory page because only the messages sent to vertices belonging to the same vertex group are stored on the same page, and the messages are read in accordance with the vertex group unit.

3.3 Main Memory Consumption

In the proposed method, the main memory consumption for graph data and messages is reduced from the amount of all vertex graph data and messages to the amount of one set of vertex group graph data and messages. This is because all graph data and messages are maintained in non-volatile memory, and only one set of group

data is read to the main memory at one time. An additional main memory region is necessary for buffers grouping and writing messages to non-volatile memory; that size is the page size of non-volatile memory multiplied by the number of vertex groups. However, this substantially reduces the amount of total main memory consumption.

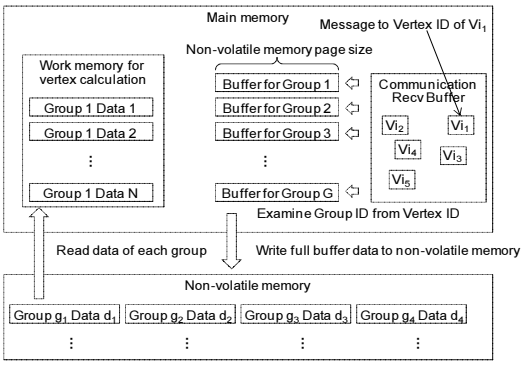

Figure 3. Illustration of proposed method.

4. IMPLEMENTATION

In our implementation of this method, a superstep on a server node is divided into stages for pipelined processing and multi-thread parallelization.

4.1 Processing Flow

The entire processing flow is shown in Figure 4. In this method, before a superstep begins, the graph data consisting of vertices and edges are read from the storage disk. The method divides vertices into the number of server nodes and sends graph data of the vertices to the assigned server node to prepare for graph processing. Then each server node groups received graph data into vertex groups and writes them to non-volatile memory in the same way as the group access method in the previous chapter.

The graph processing flow of a superstep is divided into two main parts. One involves reading data from non-volatile memory, calculating vertices, and sending messages. The other involves receiving messages and writing them to non-volatile memory.

The first part of the flow is divided into three stages. Every vertex group is processed in each stage. In the first stage, messages, the vertices, and edges in a vertex group are read from non-volatile memory. In the second stage, the read messages are sorted by a vertex identifier to gather messages with the same destination vertex identifier. In the third stage, each vertex function in a group is calculated with sorted messages, the vertices, and edges, and then it sends messages to other vertices. Then, the changed the vertices are written back to the non-volatile memory after all vertices in a group have been calculated. On the other hand, the second part of the flow is not divided into stages. Instead it receives messages in turn, groups them, and writes them to non-volatile memory.

4.2 Pipeline and Multi-Thread Processing

Using non-volatile memory efficiently and the proportional performance to the non-volatile memory bandwidth need hiding access latency of non-volatile memory that is much larger than DRAM. The first flow is divided into three pipelined stages that are processed synchronously to hide access latency of non-volatile memory. We assign each flow and stage to multi-threads for parallel computation. Because the load of each flow and stage is different, pipelined stages are synchronized. The number of threads assigned to them is regulated so that the pipeline flow is as smooth as possible.

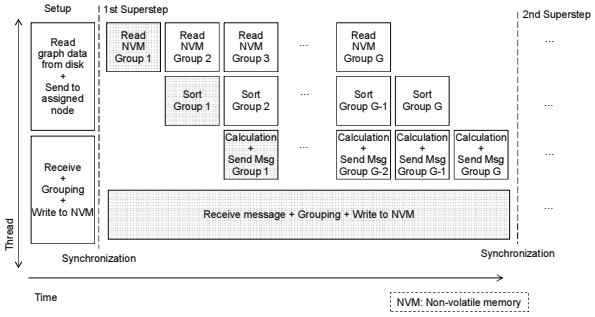

Figure 4. Proposed pipelined graph processing flow.

5. EVALUATION

In the evaluation, we confirmed the following. First, the proposed method uses SSD bandwidth efficiently and has high-performance on large-scale graph processing. Second, the main memory consumption of the proposed method is much less than the program storing all data in the main memory. Third, the proposed method and implementation hide the latency of non-volatile memory and achieve the proportional performance to the non-volatile memory bandwidth; proposed method and implementation with sufficient non-volatile memory bandwidth have not much performance loss compared with the in memory graph processing. Lastly, the proposed method has scalable performance for non-volatile memory bandwidth and for the number of nodes.

5.1 Evaluation Environment

The components used in the evaluation are listed in TABLE I (for single node evaluation) and TABLE II (for multiple nodes evaluation). The graph we used for the evaluation was constructed with the Recursive Matrix (R-MAT) [8] scale-free graph generation algorithm. The generated graphs were non-directed graphs, and their average degrees were 32. We implemented a program that carries out PageRank processing using the proposed method (NVM PageRank). For comparison, we created a PageRank program that carries out processing while storing all data including messages and graph data in the main memory (DRAM PageRank).

5.2 Non-Volatile Memory Emulator

We constructed a non-volatile memory emulator to evaluate the proposed method with many non-volatile memory parameters.

The non-volatile memory emulator imitates the memory structure shown in Figure 5. It has several buses to which non-volatile memory chips are connected. The chips in the same bus can read, program, and erase data simultaneously, but the controller cannot move the data between the memory chips and the controller simultaneously. The chips connected to different buses can work in parallel.

The emulator parameters are the number of buses, the number of chips on a bus, read time, program time, transfer speed of the memory buses, and access latency between a host program and controller. In the evaluation, the parameters are as follows: 8

chips on a bus, 50 usec read time, 1 msec program time, 96 MB/s bus speed, and 5 usec access latency. Non-volatile memory bandwidth was controlled by the number of buses.

The emulator works as follows. First, the host program writes a request to a request queue. The emulator reads the request from the request queue and records the time it reads the request. It calculates the access time on the basis of the request information and parameters of the non-volatile memory emulator. Then, it calculates the request completion time from the request start time, calculated access time, and access latency parameter. After calculating the time of the request completion, it writes the completion flag address that was specified in the request. The host program detects the completion of the request by checking this completion flag.

TABLE I. COMPONENTS USED IN SINGLE NODE EVALUATION

Hitachi HA8000 RS440AL	
CPU	Intel Xeon E7-4807 1.87 MHz (6 cores) x 4 sockets
Main memory	DDR3-1066 224 GB
SSD	Fusion-io ioDrive 80GB SLC Fusion-io ioDrive2 1.2TB MLC

TABLE II. COMPONENTS USED IN MULTIPLE NODES EVALUATION

Hitachi HA8000 RS110KML	
CPU	Intel Xeon E5-2420 1.90 MHz (6 cores) x 1 socket
Main memory	DDR3-1600 48 GB
Interconnect	Mellanox MCX3354A-QCBT (QDR x 4)
Infiniband switch	Mellanox MISS530Q-1SFC

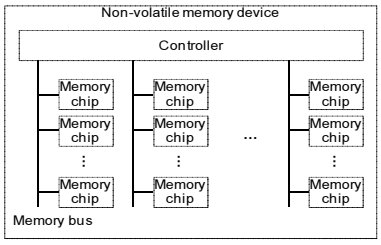

Figure 5. Non-volatile memory structure of implemented emulator.

5.3 Evaluation on SSD

5.3.1 SSD Bandwidth

In the NVM PageRank program, SSD read request IO size is 1MB and SSD write request IO is the same size as the NVM write buffer, which can be changed. For preparations of group access method evaluation, we measured SSD (ioDrive and ioDrive2) read/write mixture bandwidth on the same read/write ratio as NVM PageRank and write request IO size 8, 16, 32, 64 KB because the graph processing performance (edges per second) accorded with SSD bandwidth. The results are in TABLE III. ioDrive bandwidth improved as write IO size became big. Moreover, ioDrive2 had more intense bandwidth change caused by the write IO size than ioDrive.

TABLE III. SSD BANDWIDTH

Write IO	8KB	16KB	32KB	64KB
ioDrive	603 MB/sec	689 MB/sec	734 MB/sec	778 MB/sec
ioDrive2	641 MB/sec	951 MB/sec	1064 MB/sec	1240 MB/sec

5.3.2 Performance on a Single Node

Figure 6 plots the performance of the NVM PageRank on a single node when the numbers of vertices range from 2^{21} to 2^{25} (ioDrive) / 2^{28} (ioDrive2) and write IO size are 8, 16, 32, and 64KB. Figure 7 shows the comparison of performance between NVM PageRank and Giraph out-of-core (Giraph function of using external memory) with ioDrive / ioDrive2 on 2^{22} vertices and write IO size 64KB.

For the pipeline to work smoothly, we assigned threads to each flow and stage as follows: one thread was assigned to reading from non-volatile memory, and 20% of remaining threads were distributed to sorting messages, 40% to calculating the vertex functions, and 40% to receive/writing messages.

Performance on a single node was mostly maintained when the number of vertices was changed, and scarcely any overhead was caused as the number of vertices increased. Performance improved when the ioDrive read/write bandwidth (according with write IO size) was increased. In these cases, the SSD access speed on graph processing is from 73 to 100% of ioDrive bandwidth. Thus, the group access method uses SSD efficiently.

Giraph [4] implements the Pregel model like our proposed method and has the function of using external memory called out-of-core. We compare the performance of NVM PageRank with Giraph out-of-core with ioDrive / ioDrive2 on 2^{22} vertices and write IO size 64KB in Figure 7. About the performance of giraph out-of-core, there is scarcely any difference between ioDrive and ioDrive2. It seems that there is some bottleneck except for SSD bandwidth and it can't use SSD efficiently. On the other hand, the performance of NVM PageRank is proportional to SSD bandwidth (the detailed evaluation is on the section V.D.2).

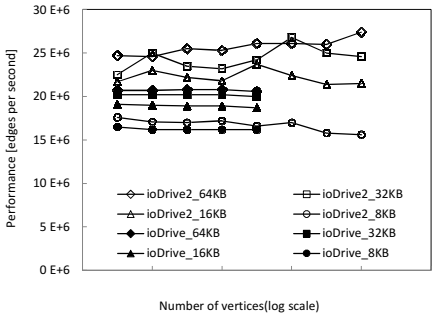

Figure 6. Performance of the NVM PageRank program.

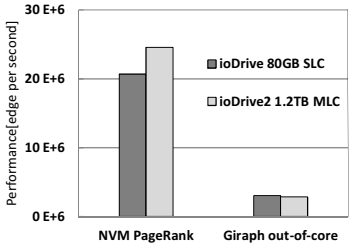

Figure 7. Comparison of performance between NVM PageRank and Giraph out-of-core

5.3.3 Memory Consumption

We evaluated the main memory and non-volatile memory consumptions of NVM and DRAM PageRank programs. Figure 8 plots the main memory and non-volatile memory consumption of the NVM PageRank program and the main memory consumption of the non-volatile memory and DRAM PageRank program when the number of vertices range from 2^{20} to 2^{27} (DRAM PageRank) / 2^{28} (NVM PageRank) on a single server node. The non-volatile memory consumption of the NVM PageRank program is almost the same as the main memory consumption of the DRAM PageRank program. This indicates that there is scarcely any overhead of the total memory amount with the proposed method. The main memory consumption ratio of this method decreases as the number of vertices increases and reaches a value of 0.8% at 2^{27} vertices. Therefore, the system with the non-volatile memory and group access method can solve 125 times bigger graph problems than the DRAM-only system. For example, the server with 32 GB DRAM and the proposed method can solve the graph processing using 8 TB non-volatile memory. The main memory consumption of NVM PageRank program is much lower than the DRAM PageRank program. This is because only one vertex group dataset is kept in the main memory on the NVM PageRank program.

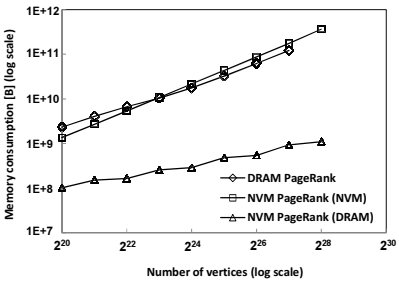

Figure 8. Main memory and non-volatile memory consumption of the NVM PageRank program and DRAM PageRank program.

5.4 Evaluation on Non-Volatile Memory Emulator

5.4.1 Performance Comparison of SSD and Non-volatile memory Emulator

Figure 9 shows the relationship between NVM PageRank graph processing performance and non-volatile memory bandwidth on a single node with ioDrive and non-volatile memory emulator. ioDrive write IO sizes are 8, 16, 32, and 64KB. The performance with ioDrive is almost the same as that with a non-volatile memory emulator at the same bandwidth. Therefore, the graph processing performance with non-volatile memory emulator is validated. In the next subsection, we evaluate the proposed method in many non-volatile memory parameters with a non-volatile memory emulator.

5.4.2 Hiding Non-Volatile Memory Access Latency

Non-volatile memory has longer access latency than DRAM. The proportional performance to the non-volatile memory bandwidth needs hiding access latency of non-volatile memory. We evaluated the relation between the performance and the access latency for examining the effect of hiding non-volatile memory latency with proposed implementation. The performance of the NVM PageRank program with 2^{25} vertices and a peak read/write bandwidth of 6.0/4.0 GB/sec on a single node when the non-volatile memory emulator access latency parameter was changed is shown in Figure 10. This implementation was able to hide access latency and the performance was almost maintained when the access latency is lower than 16 msec. It can hide the latency of general flash memory.

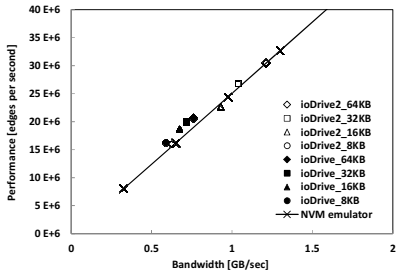

Figure 9. Relationship between NVM PageRank graph processing performance and non-volatile memory bandwidth.

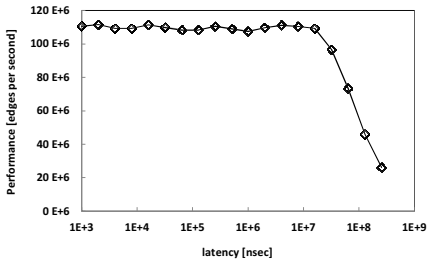

Figure 10. Relation between performance and latency.

5.4.3 Relationship between performance and bandwidth

We evaluated the relationship between the performance of the NVM PageRank program and the bandwidth of non-volatile memory in detail. Figure 11 shows the performance of the NVM PageRank program when the number of vertices was 2^{25} on a single node and the mixture read/write bandwidth was changed. The performance is proportional to the bandwidth up to the mixture read/write bandwidth of about 4.2 GB/sec. In this case, the bandwidth determines the performance, and the bottleneck is the bandwidth of non-volatile memory. Performance no longer increases even if the bandwidth is greater than 4.2 GB/sec and the performance is 107 million edges per second. This performance is 46% of the performance of DRAM PageRank (235 million edges per second). In this case, the bandwidth of non-volatile memory is sufficient, and the processing on the host server side is a bottleneck.

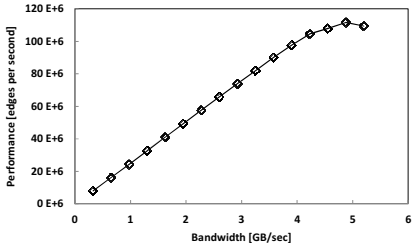

Figure 11. Relationship between performance of the NVM PageRank program and non-volatile memory mixture bandwidth.

5.4.4 Performance on Multiple Nodes

To examine the scalability of the number of server nodes, we evaluated the performance of the NVM PageRank program on multiple nodes. Figure 12 shows the performance of the NVM PageRank program on multiple server nodes when the number of vertices per node was 2^{21} and the read/write mixture bandwidth of the non-volatile emulator was 5.2 GB/sec. The performance is proportional to the number of nodes, and the performance per node is maintained when the number of nodes is changed. Thus, we confirmed that the proposed method has scalable performance for any number of nodes.

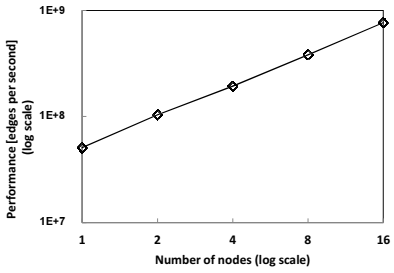

Figure 12. Performance of the NVM PageRank on multiple nodes.

6. RELATED WORK

There are some graph framework approaches to accomplish efficient graph processing with external memory. Giraph [4] has the function of using external memory called out-of-core. There is no research article about the detail of this function, but it seems that the released code implementation can't use SSD efficiently on our evaluation. GraphChi [9] proposed disk based graph processing that solved the random access problem of external memory. However, its target system is a single-node shared-memory machine, so the method is difficult to simply apply to a multi-node cluster system, unlike our proposed method. X-stream [10] also solved the random access problem of external-memory on a single-node shared-memory machine. It adopted an edge-centric model rather than a vertex-centric model used in our proposed method. On the edge-centric model, each vertex state is accessed many times on a superstep. It becomes overhead in the case of large vertex state. Pearce et al. [11][12] proposed an asynchronous system for graph traversals on external and semi-external memory. Their solution stores the read-only graph structure on disk. Vertices updated in every calculation are stored in memory, unlike our proposed method. Thus, our proposed method can process larger graph than their system because it stores the vertices in external memory and requires fewer main memory usages. In addition, their system is designed for asynchronous graph traversals, which limit applicable graph processing.

7. CONCLUSION

For high-performance large-scale graph processing, we proposed BSP graph processing in distributed non-volatile memory. It reduces main memory consumption by putting all large-scale data in non-volatile memory. By doing this, accessing non-volatile memory is a performance bottleneck because accessing messages in non-volatile memory is fine-grained random accesses and non-volatile memory has much larger latency than DRAM. Thus, we propose non-volatile memory group access method and the implementation. Proposed method and implementation improve the access performance to non-volatile memory by changing fine-grained random accesses to random accesses the same size as a non-volatile memory page and hiding non-volatile memory latency with pipelining.

We implemented the PageRank processing program with the proposed method and implementation; and evaluated it. The proposed graph processing can hide the latency of non-volatile memory and has the proportional performance to non-volatile memory bandwidth. When non-volatile memory read/write mixture bandwidth is 4.2 GB/sec, the performance of proposed graph processing and the performance storing all data in main memory have the same order of magnitude: the proposed graph processing performance is 46% of the performance of in-memory. In addition, the proposed method had scalable performance for the number of nodes. The main memory consumption of the proposed method was 0.8% total data size. Therefore, the system with non-volatile memory and the proposed method can process 125 times bigger graph than the DRAM-only system. For example, the server with 32 GB DRAM and the proposed method can process the graph using 4 TB non-volatile memories.

Overall, we have shown that our proposed method executes high-performance large-scale graph processing in a small-scale cluster system with non-volatile memory and small main memory that has reasonable cost and power consumption.

8. REFERENCES

[1] Apache Hadoop, hadoop.apache.org.

[2] Apache Spark, spark.apache.org

[3] Leiser and G. Czajkowski, "Pregel: a system for large-scale graph processing," in Proc. ACM/SIGMOD International Conference on Management of Data (SIGMOD), Indianapolis, USA, June 6–11, 2004.

[4] Apache Giraph, giraph.apache.org.

[5] 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.

[6] GraphX, spark.apache.org/graphx

[7] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999–66, Stanford InfoLab, November 1999.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in Fourth SIAM International Conference on Data Mining, April 2004.

[9] A. Kyrola, G. Glelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just aPC," in Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Hollywood, USA, October 8–10, 2012.

[10] A.Roy, I.Mihailovic, and W.Zwaenepoel, "X-Stream: edge-centric graph processing using streaming partitions," in Proc. 24th ACM Symposium on Operating Systems Principles (SOSP), Pennsylvania, USA, Nov 3–6, pp. 472–488, 2013.

[11] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory," in Proc. the International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing), New Orleans, USA, Nov. 13–19, 2010.

[12] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory," in Proc. 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Boston, USA, May 20–24, 2013.