2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Hybrid BFS Approach Using Semi-External Memory

Keita Iwabuchi*‡, Hitoshi Sato*‡, Ryo Mizote*‡, Yuichiro Yasui†‡, Katsuki Fujisawa†‡ and Satoshi Matsuoka*‡
Email: iwabuchi.k.ab@m.titech.ac.jp, hitoshi.sato@gsic.titech.ac.jp, mizote.r.aa@m.titech.ac.jp,
{yasui, fujisawa}@indsys.chuo-u.ac.jp, matsu@is.titech.ac.jp
*Tokyo Institute of Technology, Tokyo, Japan
†Chuo University, Tokyo, Japan
‡ Japan Science and Technology Agency

***Abstract*—NVM devices will greatly expand the possibility of processing extremely large-scale graphs that exceed the DRAM capacity of the nodes; however, efficient implementation based on detailed performance analysis of access patterns of unstructured graph kernel on systems that utilize a mixture of DRAM and NVM devices has not been well investigated. We introduce a graph data offloading technique using NVMs that augment the hybrid BFS (Breadth-first search) algorithm widely used in the Graph500 benchmark, and conduct performance analysis to demonstrate the utility of NVMs for unstructured data. Experimental results of a Scale27 problem of a Kronecker graph with 2^{27} vertices and 2^{31} edges compliant to the Graph500 benchmark show that our approach maximally sustains 4.22 Giga TEPS (Traversed Edges Per Second), reducing DRAM size by half with only 19.18% performance degradation on a 4-way AMD Opteron 6172 machine heavily equipped with NVM devices. Although direct comparison is difficult, this is significantly greater than the result of 0.05 GTEPS for a SCALE 36 problem by using 1TB of DRAM and 12 TB of NVM as reported by Pearce et al. [1] Although our approach uses higher DRAM to NVM ratio, we show that a good compromise is achievable between performance vs. capacity ratio for processing large-scale graphs. This result as well as detailed performance analysis of the proposed technique suggests that we can process extremely large-scale graphs per node with minimum performance degradation by carefully considering the data structures of a given graph and the access patterns to both DRAM and NVM devices. As a result, our implementation has achieved 4.35 MTEPS/W(Mega TEPS per Watt) and ranked 4th on November 2013 edition of the Green Graph500 list in the Big Data category [2] by using only a single fat server heavily equipped with NVMs.**

***Keywords*-Breadth-first search; Graph algorithms; NVM; Memory architecture; Extreme Big Data;**

I. Introduction

Recent emergence of extremely large-scale graphs in various application fields, such as health care, system biology, social networks, business intelligence, and electric power grids, etc., requires fast and scalable analysis. For example, a friend network in an existing social network service[3] is expressed as a graph with over 900 million vertices and over 100 billion edges. Rapidly increasing numbers of these large-scale graphs and their applications cause significant attractions to the Graph500 list [4], [5], which ranks supercomputers by executing large-scale graph problems as an instance of data-intensive supercomputing applications.

On the other hand, emerging NVM (Non-Volatile Memory) devices, such as Flash, have positive aspects of inexpensive cost, high energy efficiency, and huge capacity compared with conventional DRAM devices, as well as negative aspects of low throughput and latency. These NVMs will greatly expand the possibility of processing extremely large-scale graphs that exceed the DRAM capacity of the nodes; however, efficient implementation based on detailed performance analysis of access patterns of unstructured graph kernels on systems that utilize a mixture of DRAM and NVM devices has not been well investigated.

To address the above issue, we introduce a graph data offloading technique using NVMs that augment the hybrid BFS (Breadth-First Search) algorithm [6] widely used in the Graph500 benchmark, and conduct the performance analysis to demonstrate the utility of NVMs for unstructured data.

Experimental results of a Scale 27 problem of a Kronecker graph with 2^{27} vertices and 2^{31} edges compliant to the Graph500 benchmark show that our approach maximally sustains 4.22 Giga TEPS (Traversed Edges Per Second), reducing DRAM size by half with only 19.18% performance degradation on a 4-way AMD Opteron 6172 machine heavily equipped with NVM devices. Although direct comparison is difficult, this is significantly greater than the result of 0.05 GTEPS for a SCALE 36 problem by using 1TB of DRAM and 12 TB of NVM as reported by Pearce et al. [1] ; we show that a good compromise is achievable between performance vs. capacity ration for processing large-scale graphs using higher DRAM to NVM ratio. This result as well as detailed performance analysis of the proposed technique suggests that we can process extremely large-scale graphs per node with minimum performance degradation by carefully considering the data structures of a given graph and the access patterns to both DRAM and NVM devices. As a result, our implementation has achieved 4.35 MTEPS/W(Mega TEPS per Watt) and ranked 4th on November 2013 edition of the Green Graph500 list in the Big Data category [2] by using only a single fat server heavily equipped with NVMs.

II. Graph500

The Graph500 list ranks computers by executing a set of benchmarks for large-scale graph problems. In contrast

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.189

to the Top500 list [7], which is known as a list that ranks computers by executing the Linpack benchmark as an instance of compute-intensive workloads, Graph500 adopts graph processing as an instance of data-intensive workloads. Specifically, the benchmark in Graph500 measures the time for performing BFS to a Kronecker graph [8], which models real-world networks, from randomly selected 64 start points. The detailed instructions of the Graph500 benchmark are described as follows:

- **Step1 Edge List Generation**: First, the benchmark generates an edge list of an undirected graph with $N(= 2^{SCALE})$ vertices and $M(= N \cdot edge\_factor)$ edges. Here $SCALE$ denotes a base 2 logarithm of the number of vertices N, and $edge\_factor$ denotes a parameter to represent the total number of the edges M.
- **Step2 Graph Construction**: Second, the benchmark constructs a suitable data structure, such as CSR (Compressed Sparse Row) graph format, for performing BFS from the generated edge list.
- **Step3 BFS**: Then the benchmark performs BFS to the constructed data structure to create a BFS tree. Graph500 employs TEPS (Traversed Edges Per Second) as a performance metric. Thus the elapsed time of a BFS execution and the total number of processed edges determine the performance of the benchmark.
- **Step4 Validation**: Finally, the benchmark verifies the result of the BFS tree.

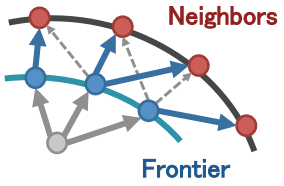
Note that the benchmark iterates Step3 and Step4 64 times, and the median value of the results is adopted as the score of the Graph500 benchmark.

III. Hybrid BFS Algorithm

We focus on the hybrid BFS approach (Hybrid BFS algorithm) [6], since the hybrid BFS algorithm can improve BFS performance drastically by reducing unnecessary edge scans. Thus most of the modern fast BFS implementations, which are ranked highly on the Graph500 list, employ the hybrid BFS algorithm. Hybrid BFS algorithm combines two approaches, a conventional *top-down approach* and a *bottom-up approach* by changing search directions. This section describes the details of these approaches and introduces how to change search directions between these two approaches.

A. Top-down Approach

The top-down approach is known as a conventional BFS algorithm. The reference code v2.1.4 of the Graph500 benchmark is also implemented by the algorithm. Figure 1 shows an outline of the top-down approach with a pseudo code. Here *frontier* denotes the set of visited vertices in a current level, *neighbors* denote the set of vertices connected to the *frontier*, and *tree* stores the visit status for each vertex in a graph. Given a current level, the top-down approach first checks each vertex v in the frontier and marks "visited (v)" if a vertex in neighbors w is marked as "unvisited (-1)". Then the vertices marked as "visited" are added to the frontier in the next level. If vertices in neighbors are marked as "visited", we skip the above operation. The top-down approach has a drawback that unnecessary vertex searches are incurred when the search state of a given graph holds many visited vertices, since the possibility of access to visited vertices from the frontier increases.



function top-down-step(frontier, next, tree)
  for $v \in$ frontier **in parallel do**
    for $w \in$ neighbors(v) **do**
      if tree$(w) = -1$ **atomic then**
        tree$(w) \leftarrow v$
        next $\leftarrow$ next $\cup\{w\}$

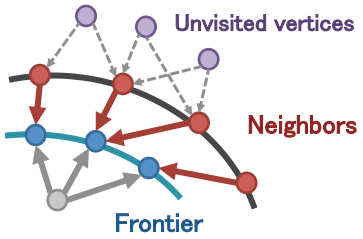
Figure 1: Outline of Top-down Approach

B. Bottom-up Approach

The bottom-up approach performs BFS in the reverse direction to the top-down approach. Figure 2 shows an outline of the bottom-up approach with a pseudo code. While the top-down approach searches for unvisited vertices from visited vertices, the bottom-up approach searches for vertices in the frontier v from all unvisited vertices w in a current level. If we find that a vertex v connected to an unvisited vertex w is included in the frontier, we mark the unvisited vertex as "visited (v)". The bottom-up approach terminates the vertex searches in a current level once we find an unvisited vertex w that is connected from a vertex v in the frontier; thus we can expect efficient BFS performance by this approach. However, the bottom-up approach also has a drawback that inefficient vertex searches are incurred when the search state of a given graph holds only a few vertices in the frontier, since possibilities of finding frontier vertices decreases.

C. Changing Search Directions

Hybrid BFS algorithm combines the benefits of the above two approaches by changing search directions. Many techniques with regard to the criterion for changing search directions has been proposed: e.g., the number of vertices in the frontier, the number of edges from the frontier, and the number of unvisited vertices, etc. [9], [6] Our hybrid BFS implementation, which is based on our proposal and is



function bottom-up-step(frontier, next, tree)
  for $w \in$ vertices **in parallel do**
    if tree$(w) = -1$ **then**
      for $v \in$ neighbors(w) **do**
        if $v \in$ frontier **then**
          tree$(w) \leftarrow v$
          next $\leftarrow$ next $\cup\{w\}$
          break

Figure 2: Outline of Bottom-up Approach

described in Section IV, changes search directions when the ratio of the number of vertices in the frontier to the total number of vertices in a given graph exceeds the predefined thresholds. Let parameters α and β be the thresholds for changing search directions, i be the current BFS level, n_{all} be the total number of vertices in a given graph, and $n_{frontier(i)}$ be the number of vertices in the frontier at the i th level. We first start BFS from a source vertex by using the top-down approach. Then, at the i th level, when $n_{frontier(i-1)} < n_{frontier(i)}$ and $n_{frontier(i)} > \frac{n_{all}}{\alpha}$, we change the top-down approach to the bottom-up approach, while when $n_{frontier(i-1)} > n_{frontier(i)}$ and $n_{frontier(i)} < \frac{n_{all}}{\beta}$, we change the bottom-up approach to the top-down approach.

IV. NUMA-optimized Hybrid BFS Implementation

We have developed a NUMA-optimized implementation of Hybrid BFS algorithm called NETAL (NETwork Analysis Library) [10], which achieves 10.5 GTEPS on the Graph500 list (November 2012) [4]. Our proposal for Hybrid BFS algorithm is based on the NETAL implementation. This section introduces the implementation details of NETAL and describes problems for processing large-scale graphs.

A. NETAL

NETAL implements Hybrid BFS algorithm by carefully considering the NUMA architecture of the underlying system. In order to perform highly efficient localized memory access during graph traversal, NETAL holds two CSR graphs for the top-down approach, called *forward graph*, and the bottom-up approach, called *backward graph*. The details of the data structures in NETAL for the Graph500 benchmark are as follows:

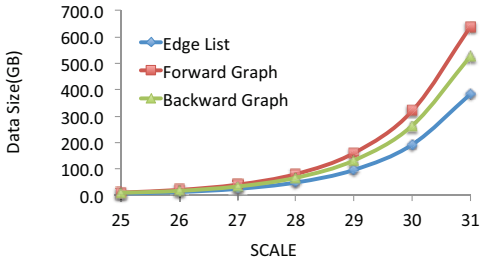


Figure 3: Breakdown of Graph Size at Each SCALE

- **Edge List**: NETAL holds an edge list of the generated Kronecker graph in a tuple format.
- **CSR Graph**: NETAL holds two CSR graphs for the top-down approach and the bottom-up approach. The top-down approach uses the *forward graph*, and the bottom-up approach uses the *backward graph*.
- **BFS Status Data**: NETAL holds data structures for BFS status management, i.e., queues, bitmaps for BFS status memories, and trees for search results.

The top-down approach explores unvisited vertices in neighbors from a vertex in the frontier. In general, each vertex straightforwardly holds all connected vertices; however, such naive graph representation introduces costly frequent memory accesses to non-local NUMA nodes. NETAL avoids inefficient memory accesses by partitioning all vertices in neighbors into a small portion of the vertices based on local NUMA nodes and duplicating corresponding frontier vertices. For example, a source vertex in a NUMA node explores destination vertices in the same NUMA node. If destination vertices belong to different NUMA nodes, NETAL delegates the search to other source vertices that belong to the same NUMA node as the destination vertices.

On the other hand, the bottom-up approach explores a vertex in the frontier from unvisited vertices. In order to provide efficient memory access to local NUMA nodes, NETAL partitions unvisited vertices based on the NUMA nodes, and candidates of vertices to the frontier are also stored on the same NUMA node.

B. Problem in BFS for Large-scale Graph

Figure 3 shows the breakdown of the graph size in NETAL. Here the x axis denotes SCALE, and the y axis denotes the actual data size. We see that the graph size exponentially increases depending on the SCALE parameter, since the graph size rises in proportion to the number of vertices. For instance, the total size of the graph, including the edge list, the forward graph, and the backward graph, at SCALE 31 reaches 1.5 TB, in which the edge list occupies 384 GB, the forward graph 640 GB, and the backward graph 528 GB. Note that the size of the forward graph exhibits slightly higher memory occupancy than that of the backward graph. In general, all graph data are stored on DRAM for

BFS execution; however, processing such extremely large-scale graphs requires huge DRAM capacity that would incur high cost and energy consumption.

V. Graph Offloading to Semi-External Memory

Emerging NVM devices, such as flash, have positive aspects of inexpensive cost, high energy efficiency, and huge capacity compared with DRAM devices, as well as negative aspects of low throughput and latency. These NVMs will greatly expand the possibility of processing extremely large-scale graphs that exceed the DRAM capacity of the nodes. As described in Section II, the Graph500 benchmark consists of 4 steps: Edge List Generation, Graph Construction, BFS, and Validation, while, as described in Section IV, the NETAL implementation for Hybrid BFS algorithm requires data structures, such as the edge list, the forward graph, the backward graph, and the BFS status data. The intuitive idea of our approach is that infrequent accessed data are offloaded to secondary semi-external memory devices such as NVMs and read the data directly from the secondary devices on demand. This section introduces the details of our graph offloading technique to semi-external memory based on the NETAL implementation.

A. Overview

In Hybrid BFS algorithm, most of visited vertices during BFS execution are referenced by the bottom-up approach, while few visited vertices are referenced by the top-down approach [9]. Thus, in order to avoid significant performance degradation due to accesses to slow devices, we offload the forward graph in the top-down approach from DRAM to NVM and read the offloaded forward graph data directly during the top-down approach. On the other hand, in the bottom-up approach, each unvisited vertex still holds connected vertices in a continuous region of the NUMA node, so that we can achieve extremely fast vertex search by utilizing local accesses in the NUMA node. Moreover, the backward graph basically includes many unreferenced vertices, since the approach terminates once a vertex in the frontier is found from unvisited vertices. Thus we can further reduce the consumption of DRAM usage of the backward graph by holding frequent accessed vertices on DRAM and offloading other vertices to NVM. Figure 4 shows an overview of our graph data offloading technique. More detailed instructions are as follows:

- **Step1 Generating the Edge List**: First, we generate an edge list on DRAM as the same way as the original Graph500 benchmark, and offload the generated edge list onto NVM.
- **Step2 Graph Construction**: We construct the forward graph on DRAM by directly reading the edge list from NVM, and offload the constructed forward graph to NVM. We also construct the backward graph on DRAM by directly reading the edge list from NVM.

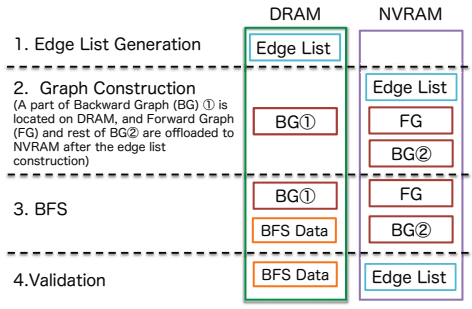


Figure 4: Overview of Our Approach

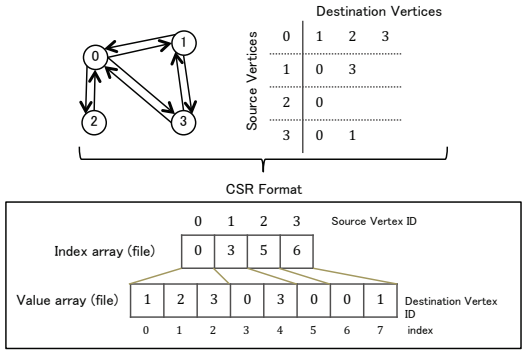


Figure 5: CSR Graph Format

Note that the constructed backward graph is basically kept on DRAM and a part of the backward graph can be offloaded to NVM for performance optimization.

- **Step3 BFS**: Before BFS execution, we create BFS status data, such as queues, bitmaps, and trees, for status management. After creating graphs and data structures, we perform BFS by using the forward graph, the backward graph, and the BFS status data.
- **Step4 Validation**: Finally, we validate the result of the BFS execution by using the BFS tree in the BFS status data on DRAM and the edge list on NVM.

Again, Step3 and Step4 are iterated 64 times.

B. Data Structure

1) CSR Graph: As described in Section IV, NETAL requires two graph data structures, called the forward graph and the backward graph, both of which are represented by the CSR graph format. Figure 5 shows an overview of the CSR graph format. The CSR graph format consists of two array structures called the *index* array and the *value* array. The *index* array stores indices of the *value* array, and the *value* array stores vertex IDs. More specifically,

each index in the *index* array represents a source vertex, and the corresponding element in the *index* array refers to an index in the *value* array. The *value* array stores IDs for destination vertices. The index in the *value* array referenced from the *index* array represents the first position of destination vertices in the edges for the source vertex. The last position of destination vertices in the edges for the source vertex is calculated from the next element for the next source vertex in the *index* array as the same way as the above instruction. In NETAL, the size of the *index* array requires the same as the number of source vertices, and the size of the *value* array requires to be twice of the number of edges, since we currently target undirected graphs which are used for the Graph500 benchmark. Our approach stores the *index* array and the *value* array on NVMs as the corresponding files respectively, which we call the *array* file and *value* file. When we read the CSR graph, our current implementation reads a continuous region for a vertex at 4KB chunks by using POSIX read(2) API.

2) Graph Partitioning Based on NUMA Node: In NETAL, vertices in both of graphs, the forward graph and the backward graph, are divided by considering NUMA nodes of the underlying system. Let n be the number of vertices, v_i be a vertex with $i \in \{0, \cdots, n-1\}$ as a vertex ID, ℓ be the number of NUMA nodes in the system, and N_k be the k th NUMA node with $k \in \{0, \cdots, \ell - 1\}$. In the above setting, we assign a vertex v_i in $i \in \left[k \cdot \frac{n}{\ell},\ (k+1) \cdot \frac{n}{\ell}\right)$ to a NUMA node N_k, so that threads assigned on the NUMA node can achieve extremely fast read operations by localizing the vertex accesses in the NUMA node. Furthermore, write operations to data structures during BFS execution, such as a bitmap that records visited vertices and an array that represents a BFS tree, are also localized in the NUMA node, since we can uniquely determine the target vertices to search in the same NUMA node. Figure 6 shows an instance of the vertex assignment to the NUMA nodes for both forward and backward graphs. In the forward graph, vertices in neighbors are divided based on the NUMA node, and vertices in the frontier are duplicated across the NUMA node. These vertices are stored in the *index* and *value* files as a single CSR graph. Note that our approach actually requires twice as many files as the number of NUMA nodes. On the other hand, in the backward graph, unvisited vertices to search are straightforwardly divided based on the NUMA nodes and stored in the corresponding CSR graphs.

C. Vertex Search

In the top-down approach, all threads running on the system are assigned to the corresponding core on a NUMA node, and each thread dequeues a fixed number (64 in our current implementation) of vertices to search neighbor vertices. Then each thread reads an element in the *array* file and calculates the position in the *value* file, then reads the *value* file in a max chunk size 4KB. We can reduce access

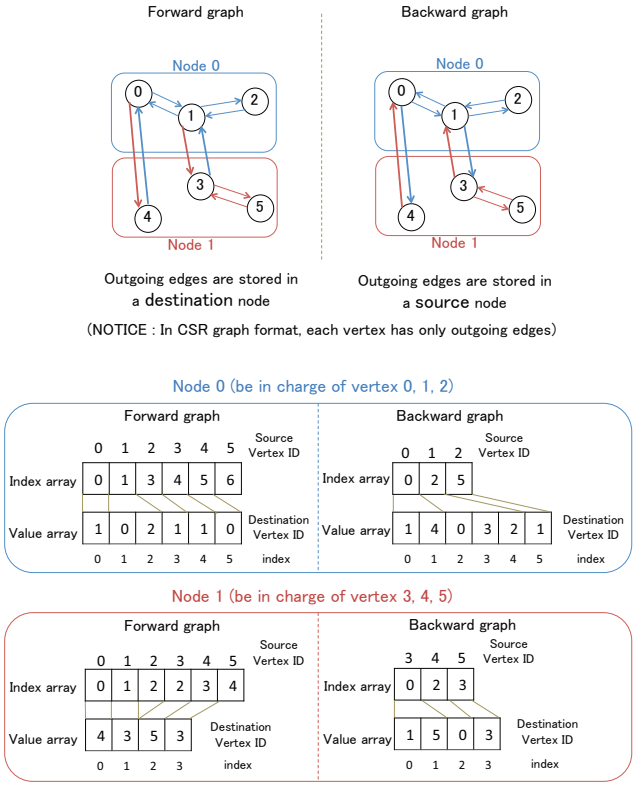


Figure 6: Overview of Vertex Assignment to NUMA Nodes

latency by sequentially reading the data.

On the other hand, in the bottom-up approach, all vertices are stored on DRAM in our current implementation. All threads are assigned to vertices that are divided based on the NUMA nodes. Then each thread searches vertices in the frontier from unvisited vertices. Although unsupported in our current implementation, we can further reduce DRAM usage if we store only a part of vertices on DRAM and offload the rest of vertices on NVM, we first read vertices on DRAM, and then we continue to read vertices on NVM in a streaming fashion.

VI. Experiments

In order to evaluate our proposed graph data offload technique described in Section V, we conduct performance analyses to demonstrate the utility of NVMs for unstructured data.

A. Experimental Setup

We set the following scenarios to emulate a node with different DRAM and NVM configurations.

- **DRAM-only**: All datasets are stored on DRAM.
- **DRAM+PCIeFlash**: Datasets are stored on DRAM and PCI Express-attached Flash Storage by using our proposed technique.

Table I: Machine Configurations

	DRAM-only	DRAM+PCIeFlash	DRAM+SSD
CPU	AMD Opteron 6172 (12 cores) × 4 sockets		
DRAM	DDR3-1333 SDRAM DIMM		
	128 GB	64 GB	
NVM	N/A	FusionIO ioDrive2 320 GB	Intel SSD 320 Series 600 GB
OS	Debian6.0 (kernel 2.6.32)		

Table II: Graph Size (SCALE 27 Edge factor 16)

	Size
Forward Graph	40.1 GB
Backward Graph	33.1 GB
BFS Status Data	15.1 GB
Total	88.3 GB

- **DRAM+SSD**: Datasets are stored on DRAM and SSD by using our proposed technique.

The details of the machine configurations are shown in Table I, which have the same CPUs, AMD Opteron 6172 (12 cores) × 4 sockets running on Debian6.0 (kernel 2.6.32), but different DRAM and NVM capacity; 128GB of DRAM (*DRAM-only*), 64GB of DRAM and 320GB of PCI Express-attached Storage (*DRAM+PCIeFlash*), and 64GB of DRAM and 600GB of SSD (*DRAM+SSD*). We use gcc v4.4 with the -O2 option for building the binary.

In this experiment, we basically use a SCALE 27 problem with edge factor 16 in the Graph500 benchmark, which consists of a Kronecker graph with 2^{27} vertices and 2^{31} edges. Table II shows the detail of the size of each dataset in the given graph, whose total size, including a forward graph, a backward graph, and BFS status data, reaches 88.3 GB. Therefore the *DRAM-only* configuration can store all the datasets on DRAM, while the other *DRAM+PCIeFlash* and *DRAM+SSD* configurations store 48.2GB of backward graph and BFS status data in DRAM and 40.1GB of forward graph on NVM.

B. BFS Performance

As described in Section III-C, we introduce two parameters α and β to change search directions between the top-down approach and the bottom-up approach. Actually, the parameter α is used to change the top-down approach to the bottom-up approach, while the parameter β is used to change the bottom-up approach to the top-down approach. In order to investigate parameter spaces on α and β, we first conducted preliminary experiments to the *DRAM-only*, *DRAM+PCIeFlash*, and *DRAM+SSD* scenarios. Figure 7 shows the performance results in TEPS in three scenarios. In each figure, the x axis denotes the α parameter, the y axis denotes the β parameter, and the color exhibits the actual median TEPS score. We see that the *DRAM-only* scenario

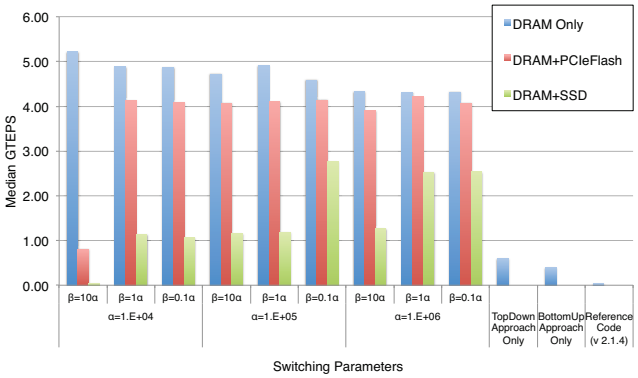
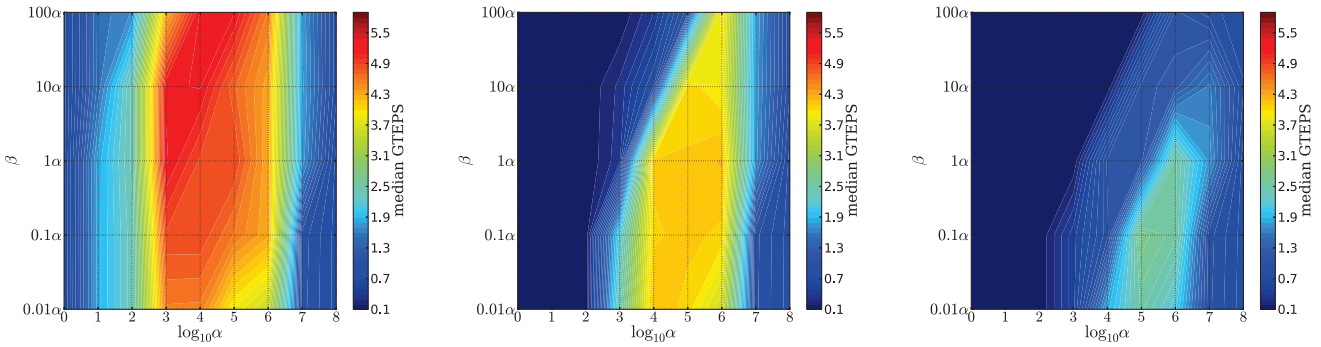


Figure 8: BFS Performance in TEPS for Kronecker Graph (Scale 27, Edge factor 16)

achieves maximally 5.12 GTEPS at $\alpha = 1.E + 04$ and $\beta = 10\alpha$, while the *DRAM+PCIeFlash* and *DRAM+SSD* scenarios achieve 4.22 GTEPS at $\alpha = 1.E + 06$ and $\beta = 1\alpha$ and 2.76 GTEPS at $\alpha = 1.E + 05$ and $\beta = 0.1\alpha$ respectively. Thus, in the following experiments, we change the α parameter from $1.E + 04$ to $1.E + 06$ and set the β parameter from $10 \cdot \alpha$ to $0.1 \cdot \alpha$ in order to compare the ratio of top-down and bottom-up approaches in Hybrid BFS algorithm.

Figure 8 shows the detailed results of the BFS performance comparison between the *DRAM-only*, *DRAM+PCIeFlash*, and *DRAM+SSD* scenarios. The x axis denotes switching parameters α and β, and the y axis denotes the median TEPS value. Figure 8 also includes the TEPS results of the top-down approach only, the bottom-up approach only, and the reference implementation of Graph500 v2.1.4 on the *DRAM-only* configuration. We see that our proposed technique exhibits good compromise; the *DRAM+PCIeFlash* scenario can achieve 4.22 GTEPS at $\alpha = 1.E + 06$ and $\beta = 1 \cdot \alpha$ with only 19.18 % performance degradation, and the *DRAM+SSD* scenario can achieve 2.76 GTEPS at $\alpha = 1.E + 05$ and $\beta = 0.1 \cdot \alpha$ with 47.1 % performance degradation. Note that the top-down only approach achieves 0.6 GTEPS, the bottom-up approach only achieves 0.4 GTEPS, and the reference implementation of Graph500 achieves 0.04 GTEPS in the same *DRAM-only* configuration.

We also evaluate our proposed technique using a smaller graph to estimate performance differences. To do so, we conduct a SCALE 26 problem with edge factor 16 on the same scenario as the above on the same configuration described in Table I. In this setting, our proposed technique requires 20.0 GB of the forward graph, 16.5 GB of the backward graph, and 10.8 GB of BFS status data, so that the required data in the *DRAM+PCIeFlash* and *DRAM+SSD* scenarios can basically be fitted on the capacity of the DRAM size. Figure 9 shows the BFS performance in TEPS



(a) DRAM-only (b) DRAM+PCIeFlash (c) DRAM+SSD

Figure 7: Average size in sectors of requests issued to NVM during BFS

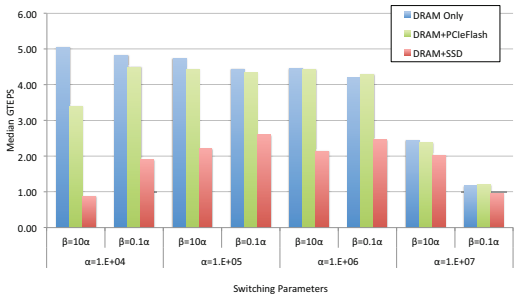


Figure 9: BFS Performance in TEPS for Kronecker Graph (Scale 26, Edge factor 16)

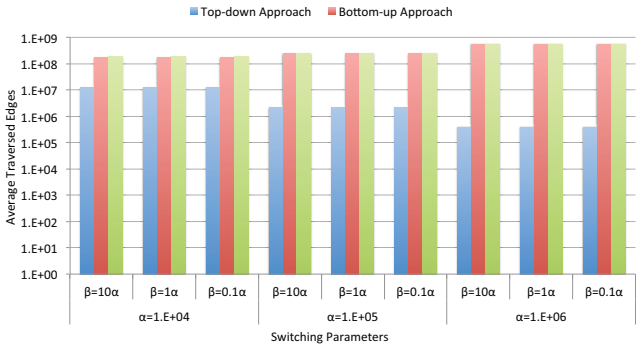


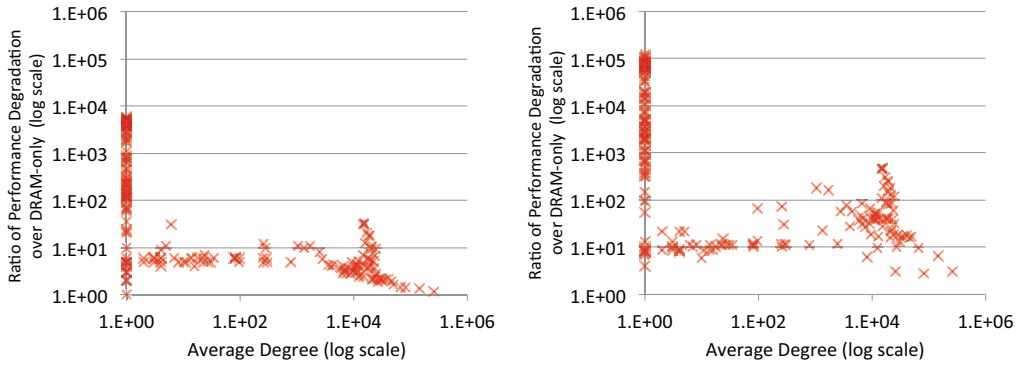
Figure 10: Traversed Edges for Kronecker Graph (Scale 27, Edge factor 16)

in each scenario. Here we see similar results to Figure 8 for the Scale 27 graph; however, the *DRAM+PCIeFlash* scenario exhibits competitive performance to the *DRAM-only* scenario. In the *DRAM+PCIeFlash* scenario, only a few top-down approaches access to the forward graph on NVM devices, and most of accesses are conducted to the backward graph on DRAM by bottom-up approaches.

C. Analysis of Performance Degradation

Next, in order to clarify the cause of the performance degradation in the previous section for a Scale27 problem, we investigate how many searches are conducted by which approaches. Figure 10 shows the comparison of the number of the average traversed edges in the top-down approach, the bottom-up approach, and the total of these approaches during the BFS execution. In our proposed technique, we minimize top-down approaches for reducing unnecessary accesses to slow NVM devices. Meanwhile, we also avoid the drawback of the bottom-up approach, which introduces inefficient vertex searches when we have a few vertices in the frontier. Choosing appropriate switching parameters in our Hybrid approach with graph data offloading technique, we can restrain performance degradation of the BFS execution.

We also investigate how our proposed technique introduces performance overheads depending on the number of edges to search for a vertex in a single level. Figure 11 shows the ratio of performance degradation in the top-dawn approach in the *DRAM+PCIeFlash* and *DRAM+SSD* scenarios compared with the *DRAM-only* scenario as a baseline. Here we set the parameters to $\alpha = 1.E+04$ and $\beta = 10 \cdot \alpha$. The x axis denotes the average number of edges to search for a vertex in a single level (average degree) in logarithmic scale, and the y axis, the ratio of performance degradation in logarithmic scale compared with the *DRAM-only* scenario. The results indicate that PCIeFlash devices achieve lower performance degradation than SSDs; the *DRAM+PCIeFlash* scenario exhibits maximally 5758.5 times and minimally 1.2 times performance degradation compared with the *DRAM-only* scenario, while the *DRAM+SSD* scenario exhibits maximally 123482.6 times and minimally 2.8 times performance degradation. We also observe that significant performance degradations occur in both NVMs when the average degrees are close to 1. This is caused by massive amounts of searches



(a) DRAM+PCIeFlash (b) DRAM+SSD

Figure 11: Ratio of performance degradation in top-down approaches compared with *DRAM-only*

from a vertex with few edges, whose situation introduces significant access concentrations to NVMs.

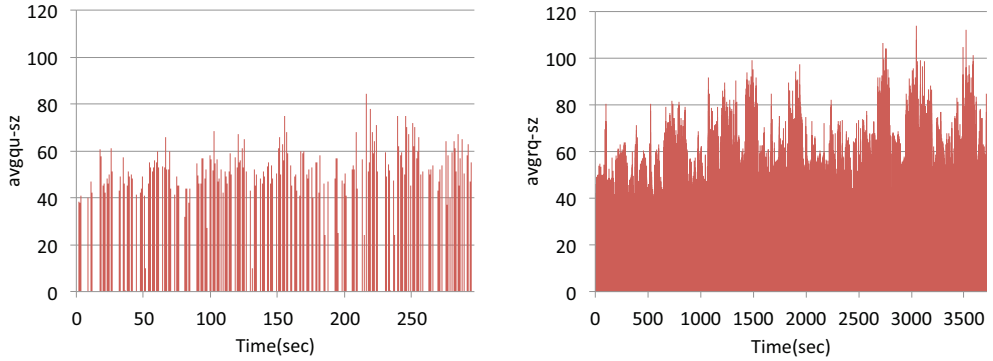
In general, during BFS execution, first several levels are conducted by top-down approaches. Then, by changing search direction using the parameter α, next several steps are conducted by bottom-up approaches. Finally, by changing search direction again using the parameter β, last several steps are conducted by top-down approaches. We also investigate the average degree during each top-down approach from Figure 11. The results show that first top-down approaches search vertices with 11182.9 degree on average, while last top-down approaches search vertices with 1 degree on average. Thus most of the searches to vertices with few degree are conducted in last several top-down approaches; however, as described above, such situation introduces significant performance degradation to access the data on NVMs. On the other hand, the bottom-up approach performs inefficient searches in the last several steps since the small number of vertices exists in the frontier. Therefore, we see one or two more additional bottom-up searches in the last several steps in the DRAM+PCIeFlash and DRAM+SSD scenarios compared with the DRAM-only scenario. Our approach can relax inefficient accesses to slow devices and process large-scale graphs with minimum performance degradation.

D. Analysis of Device Usage

In addition to the above results, we investigate the utilization of the NVM devices while performing BFS on the given graph of a Scale 27 problem by collecting I/O status via the iostat command. Figure 12 and Figure 13 show the results of the I/O status during the Graph500 benchmark execution, including iterative BFS and Validation phases 64 times. In this experiment, the CSR Graph datasets and the Edge List data, required for BFS and Validation phases, are isolated on different devices, so that Figure 12 and Figure 13 only exhibit the I/O status derived from the BFS execution. Figure 12 shows the result of *avgqu-sz* indicating the average queue length of the requests that were issued to NVMs during the BFS execution. We see that the *avgqu-sz* value achieves 36.1 in the *DRAM+PCIeFlash* scenario and 56.1 in the *DRAM+SSD* scenario on average. The results indicate that many I/O request wait situations occur during the BFS execution; however, this situation may be relaxed by using devices that achieve higher IOPS performance, since such devices can instantly evacuate I/O requests in a I/O queue. On the other hand, Figure 13 shows the result of *avgrq-sz* indicating the average size in sectors of the requests that were issued to NVMs during the BFS execution. We see that the *avgrq-sz* value achieves 22.6 in the *DRAM+PCIeFlash* scenario and 22.7 in the *DRAM+SSD* scenario on average. The results indicate that we may exploit further I/O performance of the devices by aggregating small I/O operations such as libaio library.

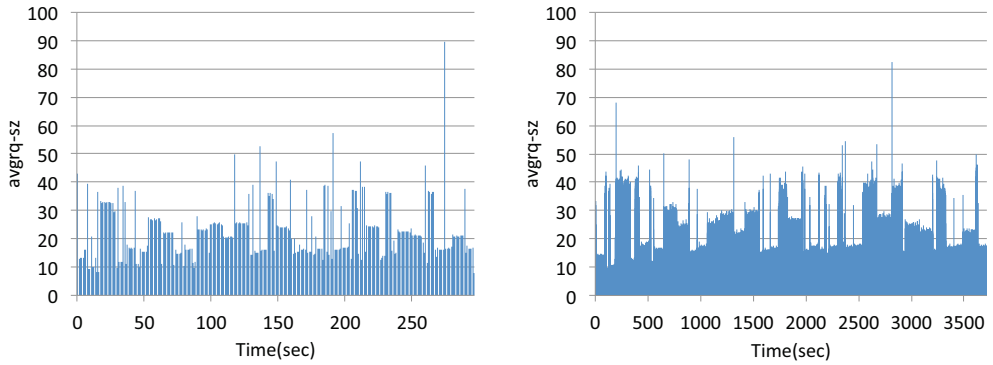
E. Towards Further Graph Data Offloading

As described in Section V, we can further reduce the consumption of DRAM usage in the bottom-up approach by holding frequent accessed vertices of the backward graph on DRAM and by offloading other vertices to NVM, since the bottom-up approach includes many unreferenced vertices due to the algorithm behavior that terminates bottom-up searches once a vertex connected from the frontier is found. In order to estimate how many vertices in the backward graph can be offloaded to semi-external memory, we investigate the accesses to NVM during the BFS execution for the given Scale 27 graph when we limit to store a part of vertices on DRAM. Figure 14 shows the access ratio to a part of the backward graph stored on NVM when we limit the number of edges for a vertex to store on DRAM. The result indicates that when we limit to store only 2 edges per vertex on DRAM, we can reduce the size of the graph on DRAM by 2.6% by storing the other edges on NVM; however, 38.2% of the edge accesses are performed on NVM. On



(a) DRAM+PCIeFlash (b) DRAM+SSD

Figure 12: Average queue length of requests issued to NVMs during BFS



(a) DRAM+PCIeFlash (b) DRAM+SSD

Figure 13: Average size in sectors of requests issued to NVM during BFS

the other hand, when we limit to store 32 edges per vertex on DRAM, we can reduce the size of the graph on DRAM by 15.1% and the accesses to the rest of the graph on NVM by 0.7%. Therefore, we can expect to offload infrequent accessed graph data onto NVMs and reduce the consumption of DRAM usage by the backward graph.

VII. Related Work

Pearce et al. has proposed NVM-based graph kernels, including BFS, for processing extremely large-scale graphs by hiding access latencies with the help of massive amounts of asynchronous multithreads [11], [12]. Pearce's algorithm can basically apply many graph algorithms over BFS; however, the algorithm requires to thoroughly scan all edges in a given graph, which introduces significant performance degradation, i.e., Pearce et al. [1] have reported the result of 0.05 GTEPS for a SCALE 36 problem by using 1TB of DRAM and 12 TB of NVM. Although direct comparison is difficult, we demonstrate a good compromise is achievable between performance vs. capacity ration for processing large-scale graphs using higher DRAM to NVM ratio.

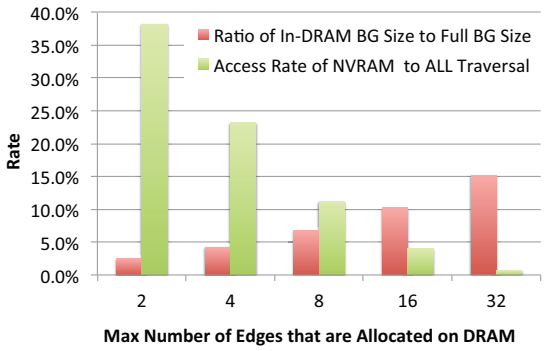


Figure 14: Access ratio to the backward graph (BG) on NVM

Kyrola et al. [13] has also proposed a latency hiding technique that divides a large graph into small chunks and sequentially reads them using a parallel sliding windows

method in order to compute large-scale graphs on a single node; however, the technique can not contribute performance improvements in Hybrid BFS algorithm, since Hybrid BFS algorithm essentially includes massive amounts of random memory access operations, especially in the top-down approach.

With regard to graph kernels to a distributed memory environment, Beamer et al. [14] has extended the hybrid BFS algorithm for multi-node systems. Pearce et al. [1] has also proposed a scaling technique of graph kernels based on NVMs.

VIII. Conclusions

NVM devices will greatly expand the possibility of processing extremely large-scale graphs that exceed the DRAM capacity of the nodes; however, efficient implementation based on detailed performance analysis of access patterns of unstructured graph kernel on systems that utilize a mixture of DRAM and NVM devices has not been well investigated. We introduce a graph data offloading technique using NVMs for Hybrid BFS algorithm based on our NETAL implementation. This results demonstrates that we can achieve 4.22 GTEPS at the maximum and reduce half the size of DRAM with 19.18% performance degradation for a SCALE 27 problem in the Graph500 benchmark. Performance analyses of our proposed implementation indicates that we can process large scale graphs with minimum performance degradation using NVM by carefully considering the data structures and the access patterns. The results also indicate that we can process large scale problem with low power consumption. As an instance, our implementation has achieved 4.35 MTEPS/W on a Huawei's 4-way system with 500 GB of DRAM and 4TB of NVMs, and ranked 4th on November 2013 edition of the Green Graph500 in the Big Data category [2] by using only a single server.

Future work includes further offloading graph data especially with small edges, performance studies on various NVM devices, and applying our technique to multi-node environments.

Acknowledgment

This work is partially supported by the JST-CREST Projects: "Advanced Computing and Optimization Infrastructure for Extremely Large-Scale Graphs on Post Peta-Scale Supercomputers" and "Extreme Big Data (EBD): Next Generation Big Data Infrastructure Technologies Towards Yottabyte/Year".

References

[1] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory," in *2013 IEEE International Parallel & Distributed Processing Symposium (IPDPS2013)*, 2013, pp. 825–836.

[2] "The Green Graph500 list." [Online]. Available: http://www.green.graph500.org

[3] "Facebook." [Online]. Available: https://www.facebook.com/

[4] "The Graph500 list." [Online]. Available: http://www.graph500.org

[5] R. Murphy, K. Wheeler, B. Barrett, and J. Ang, "Introducing the graph 500," in *Cray User's Group (CUG)*, 2010.

[6] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. Salt Lake City, USA: IEEE Computer Society Press, 2012, pp. 12:1–12:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2389013

[7] "The Top500 list." [Online]. Available: http://www.top500.org

[8] J. Leskovec and D. Chakrabarti, "Kronecker graphs: An approach to modeling networks," *The Journal of Machine Learning Research*, pp. 1–58, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756039

[9] S. Beamer, K. Asanovic, and D. Patterson, "Searching for a Parent Instead of Fighting Over Children: A Fast Breadth-First Search Implementation for Graph500," *Technical Report UCB/EECS-2011-117, EECS Department, University of California, Berkeley*, pp. 1–9, 2011.

[10] Y. Yasui, K. Fujisawa, and K. Goto, "NUMA-optimized Parallel Breadth-first Search on Multicore Single-node System," in *2013 IEEE International Conference on Big Data (IEEE BigData 2013)*, 2013.

[11] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*. IEEE, Nov. 2010, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1884643.1884675

[12] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale, "On the Role of NVRAM in Data-intensive Architectures: An Evaluation," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. Ieee, May 2012, pp. 703–714. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6267871

[13] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi : Large-Scale Graph Computation on Just a PC Disk-based Graph Computation," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, 2012, pp. 31–46.

[14] S. Beamer, A. Buluc, K. Asanovic, and D. Patterson, "Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search," in *Workshop on Multithreaded Architectures and Applications (MTAAP2013) in conjunction with 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS2013)*, Boston, 2013, pp. 1 – 10. [Online]. Available: http://www.stormingmedia.us/58/5845/A584575.html