Fast and Scalable NUMA-based Thread Parallel Breadth-first Search

Yuichiro Yasui
Kyushu University & JST COI
744 Motooka, Nishi-ku, Fukuoka, Japan
y-yasui@imi.kyushu-u.ac.jp

Katsuki Fujisawa
Kyushu University & JST CREST
744 Motooka, Nishi-ku, Fukuoka, Japan
fujisawa@imi.kyushu-u.ac.jp

***Abstract*—The breadth-first search (BFS) is one of the most centric kernels in graph processing. Beamer's direction-optimizing BFS algorithm, which selects one of two traversal directions at each level, can reduce unnecessary edge traversals. In a previous paper, we presented an efficient BFS for a non-uniform memory access (NUMA)-based system, in which the NUMA architecture was carefully considered. In this paper, we investigate the locality of memory accesses in terms of the communication with remote memories in a BFS for a NUMA system, and describe a fast and highly scalable implementation. Our new implementation achieves performance rates of 174.704 billion edges per second for a Kronecker graph with 2^{33} vertices and 2^{37} edges on two racks of a SGI UV 2000 system with 1,280 threads. The implementations described in this paper achieved the fastest entries for a shared-memory system in the June 2014 and November 2014 Graph500 lists, and produced the most energy-efficient entries in the second, third, and fourth Green Graph500 lists (big data category).**

***Keywords*—Breadth-first search; Graph algorithm; parallel algorithm; NUMA-aware; Graph500 benchmark**

I. INTRODUCTION

A. Big Data Analysis using Graph Exploration

The breadth-first search (BFS) is one of the most important graph analysis kernels. It can be used to obtain certain properties about the connections between the nodes in a given graph. BFS is not only used as a stand-alone kernel, but also functions as a subroutine in applications that determine maximum flow [13], connected components [11], centrality [6], [14], and clustering [16]. Recently, in many application fields, attempts have been made to analyze the characteristics and properties of the networks generated by social networking services [2], [20], [21], [23], [24], [25], [33]. In particular, in some recent big data projects, streams have been represented by mathematical graphs over wide areas, and these kernels have been used for further analysis[a,b]. Theoretically, BFS has a linear complexity of $O(n + m)$, where $n = |V|$ is the number of vertices and $m = |E|$ is the number of edges in a given graph $G = (V, E)$. This complexity is optimal for theoretical purposes, but there is an actual need for efficient graph processing for large-scale small-world networks on high-performance computers and/or supercomputers. Theoretical complexity analysis alone is not sufficient, because large-scale BFS computations require a significant amount of memory to enable multiple memory accesses over a wide memory space. In 2010, the Graph500 benchmark was developed. This uses graph processing to rate supercomputer systems, and is an alternative to the Top500 ranking, which uses the Linpack benchmark. In this paper, we discuss efficient graph traversal algorithms, and consider the locality of memory accesses in non-uniform memory access (NUMA) architectures.

[a]SNAP: Stanford Network Analysis Project: http://snap.stanford.edu.
[b]Human Brain Project: http://www.humanbrainproject.eu.

B. Related Work

Studies on BFS include many efficient implementations and algorithms. For a shared-memory multicore system, Agarwal et al. proposed a NUMA-aware implementation [1] of a fundamental level-synchronized BFS algorithm [3]. Beamer et al. developed a direction optimizing algorithm [4], [5] that reduces unnecessary edge traversal and is approximately five times faster than the NUMA-aware level-synchronized BFS algorithm [1]. We proposed NUMA-optimized techniques for the direction-optimizing algorithm [35], [36]. Although our NUMA-aware techniques are similar to Agarwal et al.'s proposal, there are some differences besides the use of direction. In particular, our NUMA-optimized top-down approach does not need to access remote memory in the traversal phase, and improves the locality of memory accesses. Furthermore, we adopted extension techniques for the NUMA-optimized algorithm to provide fast non-volatile memory (NVM) [18], [19]. This NVM extension allows us to relax the limitations on memory space used to migrate data to NVM from the main memory for frequency of use, similar to the method in [28]. Additionally, there are many related studies on a distributed-memory multicore multi-node system [7], [10], [29], [31], [32], [37]. The decomposition techniques used in these studies have a significant effect on our NUMA-aware implementation.

C. Contributions

In this paper, we investigate the locality of memory accesses in our BFS algorithm, and propose a fast and highly scalable thread parallel BFS algorithm for a NUMA system. Our new implementation has three variants (the *DG-V* (Dual-Graph with sparse-Vector), *DG-S* (Dual-Graph with bitmap-

and-Summary), and *SG* (Single-Graph)) for using graph representations and queue data structures for a subset of vertices. These apply the vertex sorting technique in [31] to our NUMA-optimized degree-aware implementation [36] to improve the locality of memory accesses. The differences between the variants lie in their graph representations and the queue structure used in a BFS. *DG-V* and *DG-S* use a forward graph to hold outgoing edges in the top-down direction, and a backward graph to hold incoming edges in the bottom-up direction. This reduces the number of remote memory accesses on a NUMA system. *DG-V* and *DG-S* use a sparse vector queue and bitmap queue, respectively. In contrast, *SG* uses a backward graph in both directions. In addition, the top-down algorithm of *SG* is identical to Agarwal et al.'s algorithm.

Table I lists the BFS performance of each of these implementations in terms of traversed edges per second (TEPS) and TEPS per watt (TEPS/W). *DG-V* achieves a performance of over 40 GTEPS for medium size problems (SCALE 27) on a small NUMA system (such as with four CPU sockets), whereas *DG-S* and *SG* scale up to 64 and 128 CPU sockets for large problems on a UV 2000 system, respectively. In particular, *SG* achieves 174 GTEPS on a shared-memory SGI UV 2000 supercomputer based on a cache coherent (cc)-NUMA architecture with 1,280 threads (two racks). This performance ratio of 174 GTEPS indicates that our *SG* for SCALE 34 on a UV 2000 system with 1,280 cores (128 CPU sockets) is faster than an MPI-parallel implementation [10] on IBM BG/Q with 512 nodes (8,192 cores) for SCALE 33 (172 GTEPS). Thus, our NUMA-based implementation exhibits almost the same performance as an MPI implementation. In addition, in our implementation, *DG-S* and *SG* achieved the fastest entries for a shared-memory single-node system in the June 2014 and November 2014 Graph500 lists. These variants are also highly energy-efficient, achieving first position in the small data category of the second and the big data category of the third and fourth Green Graph500 lists.

D. Graph500 and Green Graph500 Benchmarks

The Graph500 benchmark[c] is designed to measure computational performance for applications that require an irregular memory access pattern. Following its announcement in June 2010, the first Graph500 list was published in November 2010. An implementation of this benchmark must perform the following steps:

1) **Generation.** This step generates the edge list of the Kronecker graph [22] using a recursive matrix (R-MAT) computation [9]. The generator requires the *scale* and the *edgefactor* as inputs, and computes *scale* times the Kronecker products of an initiator matrix $\begin{pmatrix} 0.57 & 0.19 \\ 0.19 & 0.10 \end{pmatrix}$. The outputs are the 2^{scale} vertices and $2^{\text{scale}} \cdot \text{edgefactor}$ edges, which contain some self-loops and some parallel edges.
2) **Construction (timed).** This step constructs the graph representation from the edge list obtained in Step 1.
3) **BFS iterations (timed).** This step iterates the timed *BFS* phase and the untimed *verify* phase 64 times. The former executes the BFS for each source, and the latter confirms the output.

The Graph500 benchmark is based on the TEPS ratio, which is computed for a given graph, and the BFS output [27]. The Green Graph500 benchmark[d] is designed to measure the energy efficiency of a computer in terms of TEPS/W [17]. These lists have been updated biannually since their introduction.

II. Algorithm and Data Structure

A. Preliminaries

We assume that the input of a BFS is a graph $G = (V, E)$ consisting of a set of vertices V and a set of edges E. The connections of G are contained as pairs (v, w), where $v, w \in V$. The set of edges E corresponds to a set of adjacency lists A, where an adjacency list $A(v)$ contains the adjacency vertices w of outgoing edges $(v, w) \in E$ for each vertex $v \in V$. A BFS explores the various edges spanning all other vertices $v \in V \backslash \{s\}$ from the source vertex $s \in V$ in a given graph G, and outputs the *predecessor map* π, which is a map from each vertex v to its parent. When the predecessor map $\pi(v)$ points to only one parent for each vertex $v \in V$, it represents a tree with the root vertex $s \in V$. In addition, the predecessor map of source vertex $\pi(s)$ points itself to s.

The fundamental BFS, called level-synchronized BFS, is described in Algorithm 1. In this algorithm, we assume that an input graph $G = (V, A)$ based on an adjacency vertex list is a directed graph. If an input graph is undirected, it can be solved by replacing $(v, w) \in E$ edges with (v, w) and (w, v). The

Algorithm 1: Level-synchronized BFS

```
Input : Directed graph $G = (V, A)$, source vertex $s$, current
        queue CQ, neighbor queue NQ, and visited vertices
        VS.
Output: Predecessor map $\pi(v)$.
1  $\pi(v) \leftarrow \perp, \forall v \in V \backslash \{s\}$
2  $\pi(s) \leftarrow s$
3  VS $\leftarrow \{s\}$
4  CQ $\leftarrow \{s\}$
5  NQ $\leftarrow \emptyset$
6  while CQ $\neq \emptyset$ do
7      NQ $\leftarrow \emptyset$
8      for $v \in$ CQ in parallel do
9          for $w \in A(v)$ do
10             if $w \notin$ VS atomic then
11                 $\pi(w) \leftarrow v$
12                 VS $\leftarrow$ VS $\cup \{w\}$
13                 NQ $\leftarrow$ NQ $\cup \{w\}$
14     CQ $\leftarrow$ NQ
```

[c]Graph500 benchmark: http://www.graph500.org.

[d]Green Graph500 benchmark: http://green.graph500.org.

TABLE I: BFS Performance (TEPS Ratio and TEPS/W) of Previous Systems.

(a) Related work on single server

Reference	**Base architecture** (#CPU cores)	**RAM**	$\log_2 n$	m/n	**GTEPS**	**MTEPS/W**	**GreenGraph500**
Graph500 Reference [27]	4-way Intel Xeon E5-4640 (32)	512 GB	27	16	0.1	0.20	
Madduri et al. [3]	Cray MTA-2 (40)	–	21	512	0.5	-	
Agarwal et al. [1]	4-way Intel Xeon 7500 (32)	512 GB	20	64	1.3	-	
Beamer et al. [4], [5]	4-way Intel Xeon E7-8870 (80)	512 GB	28	16	5.1	-	
Yasui et al. [35]	4-way Intel Xeon E5-4640 (32)	512 GB	26	16	11.2	17.39	
Yasui et al. [36]	Sony Xperia-A-SO-04E (4)	2 GB	20	16	0.48	153.17	Nov. 2013 #1
	4-way Intel Xeon E5-4640 (32)	512 GB	27	16	29.0	45.43	
	SGI Altix UV 1000 (512)	512 GB	30	16	37.7	1.89	

(b) Previous systems on multi-node server

Reference	**Base architecture** (#CPU cores)	**RAM**	$\log_2 n$	m/n	**GTEPS**	**MTEPS/W**	**GreenGraph500**
Buluç et al. [7]	Hopper 1366 nodes (40000)	–	32	16	18	–	
Fabio et al. [10]	IBM BlueGeneQ 64 nodes (1024)	1 TB	30	16	29	–	
	IBM BlueGeneQ 128 nodes (2048)	2 TB	31	16	60	–	
	IBM BlueGeneQ 256 nodes (4096)	4 TB	32	16	87	–	
	IBM BlueGeneQ 512 nodes (8192)	8 TB	33	16	172	–	
	IBM BlueGeneQ 1024 nodes (16384)	16 TB	34	16	382	–	
Ueno et al. [31]	TSUBAME 2.0 1366 nodes (16392)	73.7 TB	36	16	104	–	
Ueno et al. [32]	TSUBAME 2.0 1366 nodes (4096 GPUs)	73.7 TB	35	16	317	–	

(c) System reported in this paper

Reference	**Base architecture** (#CPU cores)	**RAM**	$\log_2 n$	m/n	**GTEPS**	**MTEPS/W**	**GreenGraph500**
This paper (*DG-V*)	4-way Intel Xeon E5-4640 (32)	512 GB	27	16	41.80	87.12	
This paper (*SG*)	4-way Intel Xeon E5-4640 (32)	512 GB	30	16	28.48	59.12	June 2014 #1
	4-way Intel Xeon E5-4640 (32)	512 GB	30	16	28.61	61.48	Nov. 2014 #1
	4-way Intel Xeon E7-4890 (60)	2 TB	32	16	55.74	44.42	Nov. 2014 #3
This paper (*DG-V*)	SGI UV 2000 (640)	16 TB	31	16	124.16	–	
This paper (*DG-S*)	SGI UV 2000 (640)	16 TB	32	16	131.43	12.48	June 2014 #6
This paper (*SG*)	SGI UV 2000 (640)	16 TB	32	16	131.43	–	
	SGI UV 2000 (1280)	32 TB	33	16	174.70	–	

algorithm starts with a traversal frontier (current queue) CQ as the source s. At each level k, this algorithm finds unvisited adjacency vertices that are connected to the current frontier CQ, and appends them to the next frontier NQ for level k+1. After the edge traversal, NQ becomes the traversal frontier CQ for the next level. When the frontier is empty, the algorithm terminates.

B. Direction-optimizing Algorithm

Beamer et al. [4], [5] proposed a hybrid framework for the BFS algorithm (Algorithm 2) that reduces the number of edges explored. This algorithm combines two different traversal directions: *Top-down* and *Bottom-up*. The former traverses the *neighbors* NQ from the *frontier* CQ, whereas the latter finds the *frontier* CQ from vertices in candidate neighbors (all unvisited vertices $V\backslash$VS). This algorithm performs a traversal (lines 7–10) and swaps NQ and CQ (line 11) at each level.

Table II describes how the traversal direction is determined for *Top-down* and *Bottom-up* approaches (line 7). The traversal policy of the hybrid algorithm moves from *Top-down* to *Bottom-up* in the *growing* phase |CQ| < |NQ|, and returns from *Bottom-up* to *Top-down* in the *shrinking* phase |CQ| ≥ |NQ|. The algorithm uses the exact and approximate number of traversed edges $m_{\mathcal{F}}$ and $m'_{\mathcal{F}}$ in the *Top-down* policy, and the approximate number of traversed edges in the *Bottom-up* case, $m'_{\mathcal{B}}$. These are defined as follows:

$$m_{\mathcal{F}} \leftarrow \left|\left\{(v,w) \in E \mid v \in \text{NQ},\ w \in A^{\mathcal{F}}(v)\right\}\right|, \quad (1)$$

$$m'_{\mathcal{F}} \leftarrow |\text{NQ}| \cdot \text{edgefactor}, \quad (2)$$

$$m'_{\mathcal{B}} \leftarrow |V \setminus \text{VS}| \cdot \text{edgefactor} + |V|. \quad (3)$$

In addition, we determine the optimum values of the switching parameters α and β.

TABLE II: **is_top_down_direction** Procedure in Algorithm 2.

(a) Growing phase |CQ| < |NQ|

Current \ Next	Top-down	Bottom-up
Top-down	$m_{\mathcal{F}} \cdot \alpha < m'_{\mathcal{B}}$	$m_{\mathcal{F}} \cdot \alpha \geq m'_{\mathcal{B}}$
Bottom-up	False	True

(b) Shrinking phase |CQ| ≥ |NQ|

Current \ Next	Top-down	Bottom-up
Top-down	$m'_{\mathcal{F}} \cdot \beta < m'_{\mathcal{B}}$	$m'_{\mathcal{F}} \cdot \beta \geq m'_{\mathcal{B}}$
Bottom-up		

C. Processor and Memory Affinities

Current processor designs are based on a NUMA architecture. On such NUMA and cc-NUMA systems, each processor

Algorithm 2: Direction-optimizing BFS

```
Input  : Directed graph $G = (V, A^{\mathcal{F}},\ A^{\mathcal{B}})$, source vertex $s$,
         frontier queue CQ, neighbor queue NQ, and visited
         vertices VS.
Output: Predecessor map $\pi(v)$.
$\pi(v) \leftarrow \perp, \forall v \in V \backslash \{s\}$
$\pi(s) \leftarrow s$
VS $\leftarrow \{s\}$
CQ $\leftarrow \{s\}$
NQ $\leftarrow \emptyset$
while CQ $\neq \emptyset$ do
    if is_top_down_direction(CQ, NQ, VS) then
        NQ $\leftarrow$ Top-down($G$, CQ, VS, $\pi$)
    else
        NQ $\leftarrow$ Bottom-up($G$, CQ, VS, $\pi$)
    swap(CQ, NQ)

Procedure Top-down($G$, CQ, VS, $\pi$)
    NQ $\leftarrow \emptyset$
    for $v \in$ CQ in parallel do
        for $w \in A^{\mathcal{F}}(v)$ do
            if $w \notin$ VS atomic then
                $\pi(w) \leftarrow v$
                VS $\leftarrow$ VS $\cup \{w\}$
                NQ $\leftarrow$ NQ $\cup \{w\}$
    return NQ

Procedure Bottom-up($G$, CQ, VS, $\pi$)
    NQ $\leftarrow \emptyset$
    for $w \in V \setminus$ VS in parallel do
        for $v \in A^{\mathcal{B}}(w)$ do
            if $v \in$ CQ then
                $\pi(w) \leftarrow v$
                VS $\leftarrow$ VS $\cup \{w\}$
                NQ $\leftarrow$ NQ $\cup \{w\}$
                break
    return NQ
```

has local memory, and these connect to one another via an interconnect such as the Intel QPI, AMD HyperTransport, or SGI NUMAlink 6. On such systems, processor cores can access their local memory faster than they can access remote (non-local) memory (i.e., memory local to another processor or memory shared between processors). To some degree, the performance of BFS depends on the speed of memory access, because the complexity of memory accesses is greater than that of computation. Therefore, in this paper, we propose a general management approach for processor and memory affinities on a NUMA system. However, we cannot find a library for obtaining the position of each running thread, such as the CPU socket index, physical core index, or thread index in the physical core. Thus, we have developed a general management library for processor and memory affinities, the ubiquity library for intelligently binding cores (ULIBC). The latest version of this library is based on the HWLOC (portable hardware locality) library, and supports many operating systems (although we have only confirmed Linux, Solaris, and AIX). ULIBC can be obtained from

https://bitbucket.org/ulibc.

The HWLOC library manages computing resources using resource indices. In particular, each processor is managed by a processor index across the entire system, a socket index for each CPU socket (NUMA node), and a core index for each logical core. However, the HWLOC library does not provide a conversion table between the running thread and each processor's indices. ULIBC provides an "MPI rank"-like index, starting at zero, for each CPU socket, each physical core in each CPU socket, and each thread in each physical core, which are available for the corresponding process, respectively. We have already applied ULIBC to graph algorithms for the shortest path problem [34], BFS [18], [19], [35], [36], and mathematical optimization problems [15].

D. NUMA-optimized BFS

Our NUMA-optimized algorithm, which improves the access to local memory, is based on Beamer et al.'s direction-optimizing algorithm. Thus, it requires a given graph representation and working variables to allow a BFS to be divided over the local memory before the traversal. In our algorithm, all accesses to remote memory are avoided in the traversal phase using the following column-wise partitioning:

$$V = \left[V_0 \mid V_1 \mid \cdots \mid V_{\ell-1}\right], \tag{4}$$
$$A = \left[A_0 \mid A_1 \mid \cdots \mid A_{\ell-1}\right], \tag{5}$$

and each set of partial vertices V_k on the k-th NUMA node is defined by

$$V_k = \left\{ v_j \in V \mid j \in \left[\frac{k}{\ell} \cdot n, \frac{(k+1)}{\ell} \cdot n \right) \right\}, \tag{6}$$

where n is the number of vertices and the divisor ℓ is set to the number of NUMA nodes (CPU sockets). In addition, to avoid accessing remote memory, we define partial adjacency lists $A_k^{\mathcal{F}}$ and $A_k^{\mathcal{B}}$ for the *Top-down* and *Bottom-up* policies as follows:

$$A_k^{\mathcal{F}}(v) = \{w \mid w \in \{V_k \cap A(v)\}\}, v \in V, \tag{7}$$
$$A_k^{\mathcal{B}}(w) = \{v \mid v \in A(w)\}, w \in V_k. \tag{8}$$

Furthermore, the working spaces NQ$_k$, VS$_k$, and π_k for partial vertices V_k are allocated to the local memory on the k-th NUMA node with the memory pinned. Note that the range of each current queue CQ$_k$ is all vertices V in a given graph, and these are allocated to the local memory on the k-th NUMA node. Algorithm 3 describes the NUMA-optimized *Top-down* and *Bottom-up* processes. In both traversal directions, each local thread T_k binds to the processor cores of the k-th NUMA node, and only traverses the local neighbors NQ$_k$ from the local frontier CQ$_k$ on the local memory. The computational complexities are $O(m)$ (*Top-down*) and $O(m \cdot \text{diam}_G)$ (*Bottom-up*), where m is the number of edges and diam$_G$ is the diameter of the given graph. These complexities are the same as in the original direction-optimizing algorithm. The direction-optimizing algorithm that combines these algorithms has $O(m \cdot \text{diam}_G)$ complexity. However, the actual CPU time of this algorithm is shorter than that of a Top-down only approach for a small-world network such as a Kronecker graph.

Algorithm 3: NUMA-optimized BFS

```
Input  : NUMA node indices $k \in \{0, 1, ..., \ell - 1\}$, NUMA
         local threads $T = \{T_k\}$, directed graph
         $G = \{G_k\} = \{(V_k, A_k^{\mathcal{F}}, A_k^{\mathcal{B}})\}$, duplicated frontier
         queues $\text{CQ} = \{\text{CQ}_k\}$, visited vertices
         $\text{VS} = \{\text{VS}_k\}$, and predecessor map $\pi = \{\pi_k\}$.
Output : Neighbor queues $\text{NQ} = \{\text{NQ}_k\}$.
1  Procedure NUMA-optimized-Top-down($G, \text{CQ}, \text{VS}, \pi$)
2      $\text{NQ}_k \leftarrow \emptyset$
3      for $v \in \text{CQ}_k$ in parallel($T_k$) do
4          for $w \in A_k^{\mathcal{F}}(v)$ do
5              if $w \notin \text{VS}_k$ atomic then
6                  $\pi_k(w) \leftarrow v$
7                  $\text{VS}_k \leftarrow \text{VS}_k \cup \{w\}$
8                  $\text{NQ}_k \leftarrow \text{NQ}_k \cup \{w\}$
9      return $\text{NQ}_k$
10 Procedure NUMA-optimized-Bottom-up($G, \text{CQ}, \text{VS}, \pi$)
11     $\text{NQ}_k \leftarrow \emptyset$
12     for $w \in V_k \setminus \text{VS}_k$ in parallel($T_k$) do
13         for $v \in A_k^{\mathcal{B}}(w)$ do
14             if $v \in \text{CQ}_k$ then
15                 $\pi_k(w) \leftarrow v$
16                 $\text{VS}_k \leftarrow \text{VS}_k \cup \{w\}$
17                 $\text{NQ}_k \leftarrow \text{NQ}_k \cup \{w\}$
18                 break
19     return $\text{NQ}_k$
```

In general, when the number of NUMA nodes is set to $\ell = 1$, our algorithm and the original direction-optimization are equivalent. In this case, our algorithm retains either the forward graph $A^{\mathcal{F}}$ or the backward graph $A^{\mathcal{B}}$, because these are equivalent.

E. Degree-aware Bottom-up Algorithm

In this section, we explain our previously proposed algorithm, called *Degree-aware bottom-up*, which further reduces unnecessary edge traversals in the Bottom-up policy. The bottom-up step checks that each unvisited vertex has an adjacent vertex that connects vertices on the frontier. Therefore, this step reduces the number of unnecessary edge traversals, as an adjacency vertex that has already been visited with high probability is allocated a higher position on each adjacency vertex list. However, it is difficult to obtain the optimal ordering for the adjacency vertex list. Thus, we focus on a heuristic method for the adjacency vertices based on the out-degree $\deg_G(v)$ of each vertex $v \in V$, which is defined as follows:

$$\deg_G(v) = |\{(v, w) \in E \mid w \in V\}|. \qquad (9)$$

Table III compares the number of traversed edges for each level in the *Top-down* policy with that of the *Bottom-up* approach with both *Ascending* and *Descending* ordering. The *Ascending* and *Descending* ordering construct an adjacency vertex list $A(v)$ that is sorted by $\deg_G(w), w \in A(v)$ in ascending and descending order, respectively. For this case, the algorithm selects the Top-down approach at low and high levels 0, 1, and 6, and the Bottom-up policy for middle levels 2–5. The table shows that most of the traversed edges are concentrated in level 2, and our previous proposal "Bottom-up + Descending order" (BU/Desc) has approximately ten times fewer traversals than "Bottom-up + Ascending order" (BU/Asc) in this level. The implementation of our previous BFS was based on this result.

TABLE III: Number of Traversed Edges in a BFS for a Kronecker Graph with SCALE 27.

Level	Top-down	BU/Desc	BU/Asc
0	**22**	4,223,250,243	4,223,039,317
1	**239,930**	3,258,645,723	4,063,345,725
2	1,040,268,126	**83,878,899**	**848,743,124**
3	3,145,608,885	**19,616,130**	**19,935,737**
4	37,007,608	**139,606**	**139,868**
5	98,339	**41,846**	**41,846**
6	**260**	41,586	41,586
Total	4,223,223,170	7,585,614,033	9,155,287,203
%	100%	179.6%	216.8%

A breakdown of vertex traversal for each ordering is given in Table IV, where τ is the loop count in the Bottom-up policy (Algorithm 2 line 24), which is equivalent to the location of the adjacency vertex array for finding a frontier vertex. This clearly suggests that the ordering strategies affect the loop count. The table shows that the number of zero-degree vertices is half the total number of vertices, and the locality of the descending order method is greater than that of the ascending order approach. This feature was noted in [30].

TABLE IV: Breakdown of Vertex Traversal (Descending order) in a BFS of a Kronecker Graph with SCALE 27.

Component	0-degree	BU ($\tau = 1$)	BU ($\tau \geq 2$)	TD
#vertices	71,140,085	**60,462,127**	2,358,918	215,070
Ratio	53.00%	**45.05%**	1.76%	0.16%

Here, referencing the adjacent vertices requires many indirect accesses via an index array and an adjacency edge array of the compressed sparse row (CSR) graph. We clarified that the major bottleneck of the hybrid BFS algorithm is the edge traversal of the first adjacent vertex in the Bottom-up policy, as discussed in the previous subsection. Then, we separated the standard CSR graph into the *highest-degree adjacency vertex list* $A^{\mathcal{B}+}$ and the remaining CSR graph $A^{\mathcal{B}-}$. The highest-degree adjacency vertex $A^{\mathcal{B}+}(v)$ for each vertex v contains an adjacency vertex w, the maximum degree of which is given by:

$$A^{\mathcal{B}+}(v) = \underset{w \in A^{\mathcal{B}}(v)}{\arg\max} \{|\mathrm{d}_G(w)|\}, v \in V. \tag{10}$$

We now explain the construction of $A^{\mathcal{B}+}(v)$ and $A^{\mathcal{B}-}(v)$ for each vertex $v \in V$. First, the given adjacency vertices

$$A^{\mathcal{B}}(v) = \{w_0, w_1, ..., w_{\deg_G(v)-1}\} \tag{11}$$

are sorted by out-degree as follows:

$$\deg_G(w_0) \geq \deg_G(w_1) \geq ... \geq \deg_G(w_{\deg_G(v)-1}). \tag{12}$$

The sorted $A(v)$ is then separated as follows:

$$A^{\mathcal{B}+}(v) = \{w_0\}, A^{\mathcal{B}-}(v) = \{w_1, ..., w_{\deg_G(v)-1}\}. \tag{13}$$

This graph representation requires an additional computational overhead for sorting the adjacency vertex list, but does not require increased memory. We focus on the fact that half of the total number of vertices are zero-degree vertices. This property does not significantly affect the performance of the top-down search. However, the bottom-up search is affected, because the frontier is searched from all unvisited vertices, including zero- and nonzero-degree vertices. To avoid the access cost associated with zero-degree vertices, we propose *zero-degree vertex suppression*, which renumbers the vertex ID of all nonzero vertices during graph construction. Algorithm 4 describes the degree-aware bottom-up BFS. This uses a graph representation that separates the highest-degree adjacency vertex $A^{\mathcal{B}+}$ and the remaining adjacency vertices $A^{\mathcal{B}-}$. This algorithm separates two major loops involved in this process, namely lines 3–8 and lines 9–15.

F. Sorting Vertex Indices by Out-degree (Proposal)

In this section, we propose a vertex sorting technique that is similar to that in [31]. Our new graph representation combines our degree-aware speedup technique and a vertex sorting technique based on a NUMA-aware implementation. First, in the graph construction, our vertex sorting technique constructs vertex indices $\{0, 1, ..., n-1\}$ as follows:

$$\deg_G(v_0) \geq \deg_G(v_1) \geq ... \geq \deg_G(v_{n-1}).$$

A degree-aware graph representation is then constructed using only those vertices with non-zero degree. In Table V, our speedup techniques for a Kronecker graph are compared with SCALE 27 on a SandyBridge-EP system. The results show that our new algorithm is 1.47 ($= \frac{42.5}{29.0}$) times faster than our previous method. The entire speedup is 8.33 ($= \frac{42.5}{5.1}$) times faster than that of the original direction-optimizing algorithm. This technique improves the locality of memory accesses. More details of this are given in the next section.

Algorithm 4: Degree-aware Bottom-up BFS

```
Input  : NUMA node indices $k \in \{0, 1, ..., \ell-1\}$,
         NUMA-local threads $T = \{T_k\}$, directed graph
         $G = \{G_k\} = \{(V_k, A_k^{\mathcal{B}+}, A_k^{\mathcal{B}-})\}$, duplicated frontier
         queues $\mathrm{CQ} = \{\mathrm{CQ}_k\}$, visited vertices $\mathrm{VS} = \{\mathrm{VS}_k\}$,
         and predecessor map $\pi = \{\pi_k\}$.
Output: Neighbor queues $\mathrm{NQ} = \{\mathrm{NQ}_k\}$.
1  Procedure Degree-aware-NUMA-Bottom-up($G, \mathrm{CQ}, \mathrm{VS}, \pi$)
2      $\mathrm{NQ}_k \leftarrow \emptyset$
3      for $w \in V_k \setminus \mathrm{VS}_k$ parallel($T_k$) do
4          $v \leftarrow A_k^{\mathcal{B}+}(w)$
5          if $v \in \mathrm{CQ}_k$ then
6              $\pi_k(w) \leftarrow v$
7              $\mathrm{VS}_k \leftarrow \mathrm{VS}_k \cup \{w\}$
8              $\mathrm{NQ}_k \leftarrow \mathrm{NQ}_k \cup \{w\}$
9      for $w \in V_k \setminus \mathrm{VS}_k$ parallel($T_k$) do
10         for $v \in A_k^{\mathcal{B}-}(w)$ do
11             if $v \in \mathrm{CQ}_k$ then
12                 $\pi_k(w) \leftarrow v$
13                 $\mathrm{VS}_k \leftarrow \mathrm{VS}_k \cup \{w\}$
14                 $\mathrm{NQ}_k \leftarrow \mathrm{NQ}_k \cup \{w\}$
15                 break
16     return $\mathrm{NQ}_k$
```

TABLE V: Comparison of Our Speedup Techniques.

Implementation	SCALE	TEPS
Dir. Opt. (baseline) [4], [5]	28	5.1 G
+ NUMA Opt. [35]	27	10.9 G
+ NUMA Opt. + Degree Opt. [36]	27	29.0 G
This paper (DG-V)		
+ NUMA Opt. + Degree Opt. + Vertex Sorting	27	42.5 G

G. Implementation Variants (Proposal)

We have various implementations that use graph representations and queue data structures for a subset of vertices. Our NUMA-optimized BFS algorithm needs two types of graph representation: $G^{\mathcal{F}}$ for use with Top-down policies, called a *Forward graph*, and $G^{\mathcal{B}}$ for use on Bottom-up policies, called a *Backward graph*. This model achieves high performance on a NUMA system with a few CPU sockets. However, the memory requirements of this model are $\mathcal{O}(n+\frac{m}{\ell})$ for $G^{\mathcal{F}}$ and $\mathcal{O}(\frac{n+m}{\ell})$ for $G^{\mathcal{B}}$ for each NUMA node on an ℓ-way NUMA system. Note that the forward graph requires more memory than the backward graph, because the memory usage of the index array in the forward graph does not depend on the number of NUMA nodes ℓ.

The performance characteristics of BFS implementations, such as the memory requirements and number of memory accesses, differ for each data structure used. Our BFS implementations use vertices for the current queue CQ and next queue NQ, and visited vertices VS are represented by a *Sparse-vector* ($\mathcal{V}$), *Bitmaps* ($\mathcal{B}$), or *Bitmaps-and-summary* ($\mathcal{S}$). $\mathcal{V}$ is one of the simplest representations for a subset of vertices, and represents each vertex index using each element of the array. $\mathcal{B}$ represents the set for each vertex as one bit. If the corresponding vertices are in a set, the bit is 0; otherwise, the bit is 1. Finally, $\mathcal{S}$ holds a summary of information for 64 vertices as 1 bit. If the corresponding continuous 64 vertices are all in a set, the summary bit is 0; otherwise, the summary bit is 1. The memory usage for these data structures is t elements ($\mathcal{V}$), n bits ($\mathcal{B}$), and $\frac{n}{64}$ bits ($\mathcal{S}$), where n is the number of vertices in a given graph and t is the number of vertices in the corresponding subset. A 64-bit integer is used for the vertex index, and $8t$ bytes, $\frac{n}{8}$ bytes, and $\frac{n}{4096}$ bytes are required by each data structure, respectively.

Table VI summarizes the implementation abstractions in our previous work and this study. Here, *DG-V* achieves the most TEPS at the middle level, whereas *SG* handles higher SCALE problems on a large-scale system. Finally, *DG-S* is a combination of *DG-V* and *SG*. The results for these variants are summarized in Section IV.

TABLE VI: Implementation Variants

	Top-down			Bottom-up			
Model	G	CQ	NQ	G	CQ	NQ	VS
Bigdata2013	$G^{\mathcal{F}}$	$\mathcal{V}$	$\mathcal{V}$	$G^{\mathcal{B}}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$
ISC2014	$G^{\mathcal{F}}$	$\mathcal{V}$	$\mathcal{V}$	$G^{\mathcal{B}}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$
This paper							
DG-V	$G^{\mathcal{F}}$	$\mathcal{V}$	$\mathcal{V}$	$G^{\mathcal{B}}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$
DG-S	$G^{\mathcal{F}}$	$\mathcal{S}$	$\mathcal{S}$	$G^{\mathcal{B}}$	$\mathcal{S}$	$\mathcal{S}$	$\mathcal{S}$
SG	$G^{\mathcal{B}}$	$\mathcal{V}$	$\mathcal{V}$	$G^{\mathcal{B}}$	$\mathcal{S}$	$\mathcal{S}$	$\mathcal{S}$

III. LOCALITY ANALYSIS

This section discusses the locality of memory accesses for each BFS model on a NUMA system. We define the locality of memory accesses as the number of memory accesses for each vertex in the edge traversal of 64 BFSs of the Graph500 benchmark.

A. Access Frequency and Degree Distribution

Kronecker graphs, which are used in the Graph500 benchmark, have a power-law degree distribution. Figure 1a shows the number of accesses for each vertex in the edge traversal of each BFS required by previous algorithms: Top-down [27], Direction-optimizing [4], [5], [35], and Direction-optimizing with Degree-aware graph representation [36]. Figures 1b and 1c show the equivalent results for our new implementation. These figures represent the average of 64 BFSs for a Kronecker graph with SCALE 20. The numbers of accesses correspond to the number of memory accesses for visited vertices VS in Top-down and current frontier CQ in Bottom-up. The number of accesses in the direction-optimizing BFS is smaller than for the top-down only approach, and our degree-aware technique can extract zero-degree vertices. Furthermore, our new algorithm drastically improves the locality of vertex access from irregular memory accesses in BFS. When our degree-aware and vertex sorting are applied, the access frequency distribution for each vertex in a BFS is similar to the degree distribution of input graph. In the above, the vertex sorting technique is suitable for small-world networks or power-law degree distribution networks, such as a Kronecker graph.

IV. EXPERIMENTAL RESULTS

A. System Configuration

Table VII summarizes the system configurations discussed in this section. We compared the TEPS scores and scalability of implementation variants on a 4-way Intel Xeon server (SB-EP) and the shared-memory supercomputer UV 2000 (UV2K).

TABLE VII: System Configuration

System	**#CPUs**	**CPU name (Base Architecture)**	**RAM**
SB-EP	4	8-core Xeon E5-4640 (SandyBridge-EP)	512 GB
UV2K	256	10-core Xeon E5-4650 v2 (IvyBridge-EP)	64 TB

B. Implementations and Parameters

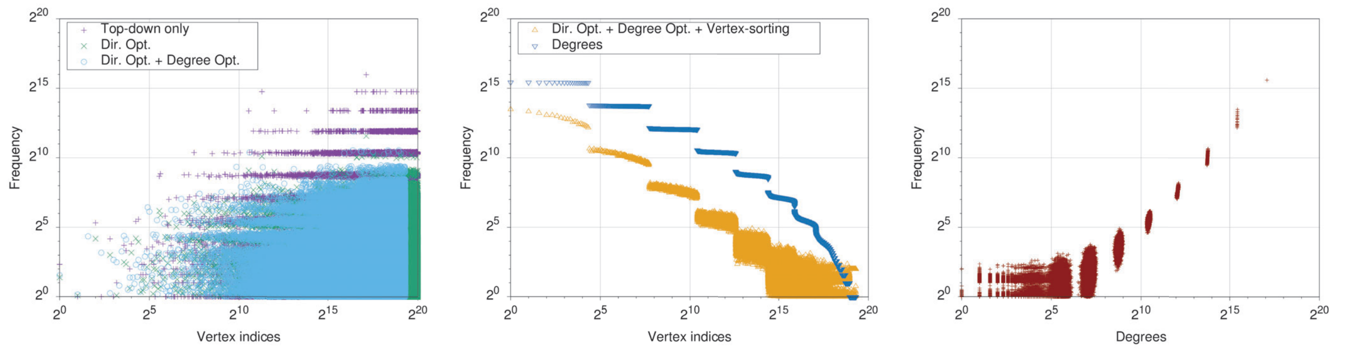
All implementations considered in this study were designed as thread parallel algorithms and coded using C with OpenMP. Our ULIBC was adopted to manage the CPU and memory affinity. We used the following switching parameters for the search direction: $(\alpha, \beta) = (16, 4)$ for *DG-V* and *DG-S*, and $(\alpha, \beta) = (512, \infty)$ for *SG*.

C. Performance Variations with Problem Size

Figure 2a compares the BFS performance of three implementation variants on the SB-EP system with 64 threads. Both *DG-V* and *DG-S* are effective up to SCALE 29, because these models need two graph representations. However, *SG* is effective up to SCALE 30, because it reduces one of the two graph representations. All models scale up to SCALE 27, but *DG-V* suffers performance degradation for large SCALEs 28 and 29. In contrast, *DG-S* and *SG* maintain their performance for large SCALEs 27, 28, and 29. The performance deterioration causes widespread memory accesses to the bitmaps array, but $\mathcal{S}$ can prevent this from occurring. However, the overall performance of $\mathcal{S}$ decreases more than that of the bitmap case.

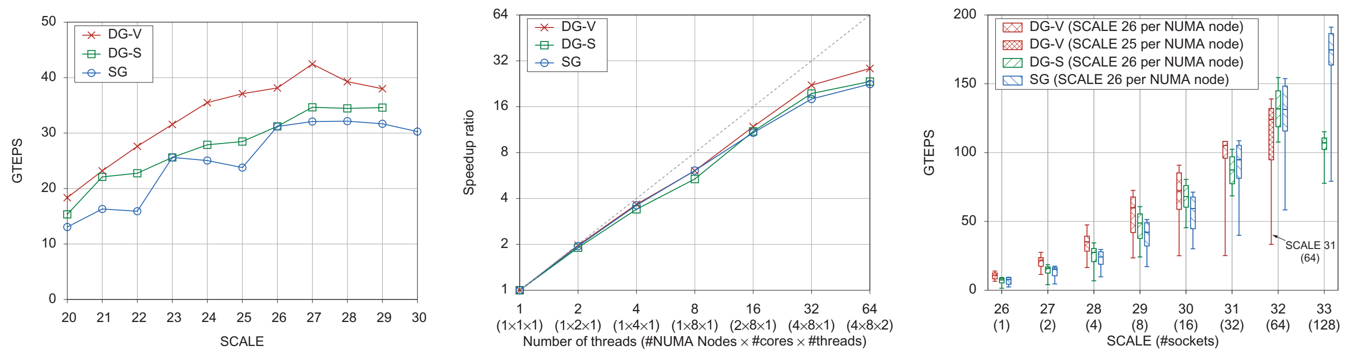
D. Strong Scaling on 4-way Intel Xeon Server

Figure 2b compares the strong scaling performance of the three implementations on the SB-EP system with SCALE 27. All implementations scale well up to 64 threads, which is equal to the number of logical cores on the system. The speedup ratios of *DG-V*, *DG-S*, and *SG* were found to be 28.5X, 23.4X, and 22.5X, respectively. Furthermore, the effect of hyperthreading (comparing the results of 32 threads and 64 threads) for the DG-V, DG-S, and SG models was to produce a speedup of 1.25X, 1.20X, and 1.25X, respectively.



(a) Previous work (b) This paper and degrees (c) Correlation between access frequency and degrees

Figure 1: Access frequency of vertex traversal for Kronecker graph with SCALE 20.



(a) TEPS with respect to problem size on SB-EP (b) Strong scaling on SB-EP (c) Weak scaling on the SGI UV 2000

Figure 2: Performance comparison of BFS models for Kronecker graph

E. Weak Scaling on SGI UV 2000

Figure 2c shows the weak scaling performance of the three implementations on the UV 2000 system with SCALE 26 per NUMA node (CPU socket). We can see that *DG-V* was faster than *DG-S* and *SG* for small problems, i.e., SCALE 26 to SCALE 31 on 1–32 sockets (320 threads; half of the UV rack). *DG-S* obtained 131.43 GTEPS for SCALE 32, exceeding the performance of *DG-V* (124.16 GTEPS for SCALE 31), on one rack of the UV 2000 system with 64 sockets (640 threads). Furthermore, *SG* scales up to 1,280 threads and achieves 174.704 GTEPS for SCALE 34. These results show that our *DG-V* is faster with fewer threads, and that *SG* scales up to thousands of threads.

V. Conclusion

In this paper, we first described the adaptation of vertex sorting techniques [31] to our previous work [35], [36]. Second, we proposed three implementation variants, *DG-V*, *DG-S*, and *SG*, for using graph representations and queue data structures for a subset of vertices, and investigated the locality of memory accesses in each of our models. Finally, we presented numerical results for each model on a 4-way Xeon server and a UV 2000 system. Our *DG-V* achieved over 40 GTEPS on the 4-way Intel Xeon server, and our *DG-S* achieved 131 GTEPS on one rack of the UV 2000 system with 640 threads. They were the most power-efficient entries of all commercial supercomputers in the June 2014 and November 2014 Green Graph500 lists. Our *SG* topped the third and fourth Green Graph500 lists (big data category). Finally, we showed that our *SG* scales up to 1,280 threads, and achieves 174 GTEPS on two racks of the UV 2000 system with 1,280 threads. This result is the fastest entry for a shared-memory single-node system in the November 2014 list of the Graph500 benchmark. In future work, we will examine a NUMA-optimized implementation for the single-source shortest path problem [8], [12], [26].

Acknowledgment

This research was supported by the Core Research for Evolutional Science and Technology (CREST) and Center of Innovation (COI) programs of the Japan Science and Technology Agency (JST), the Institute of Statistical Mathematics (ISM), and Silicon Graphics International (SGI) Corp.

References

[1] V. Agarwal, F. Petrini, D. Pasetto, and D.A. Bader. *Scalable graph exploration on multicore processors*. In ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC10) Proceedings, pp. 1–11, IEEE Computer Society, 2010.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. *Group formation in large social networks: Membership, growth, and evolution*. KDD'06 Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 44–54, 2006.

[3] D.A. Bader and K. Madduri. *Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2*. In 2006 Int. Conf. Parallel Processing (ICPP ' 06) Proceedings, pp. 523–530, IEEE Computer Society, 2006.

[4] S. Beamer, K. Asanović, and D.A. Patterson. *Searching for a parent instead of fighting over children: A fast breadth-first search implementation for Graph500*. Berkeley, CA: EECS Department, University of California, UCB/EECS-2011–117, 2011.

[5] S. Beamer, K. Asanović, and D.A. Patterson. *Direction-optimizing breadth-first search*. In ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC12) Proceedings, IEEE Computer Society, 12, 2012.

[6] U. Brandes. *A faster algorithm for betweenness centrality*. J. Math. Sociol., 25(2):163–177, 2001.

[7] A. Buluç and K. Madduri. *Parallel breadth-first search on distributed memory systems*. In ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC11) Proceedings, ACM, 65, 2011.

[8] V.T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. *Scalable single source shortest path algorithms for massively parallel systems*. In IEEE Int. Symp. Parallel and Distributed Processing (IPDPS 14) Proceedings, IEEE Computer Society, 2014.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. *R-MAT: A recursive model for graph mining*. In 4th SIAM Int. Conf. Data Mining, SIAM Proceedings, pp. 442–446, 2004.

[10] F. Checconi and F. Petrini. *Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines*. In IEEE Int. Symp. Parallel and Distributed Processing (IPDPS 14) Proceedings, IEEE Computer Society, 2014.

[11] T. Cormen, C. Leiserson, and R. Rivest. INTRODUCTION TO ALGORITHMS. MIT Press, Cambridge MA, 1990.

[12] E.W. Dijkstra. *A note on two problems in connection with graphs*. Numerische Mathematik, 1(1):269–271, 1959.

[13] J. Edmonds and R.M. Karp. *Theoretical improvements in algorithmic efficiency for network flow problems*. Journal of the ACM, 19(2):248–64, 1972.

[14] M. Frasca, K. Madduri, and P. Raghavan. *NUMA-aware graph mining techniques for performance and energy efficiency*. In ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC12) Proceedings, pp. 1–11, IEEE Computer Society, 2012.

[15] K. Fujisawa, T. Endo, Y. Yasui, H. Sato, N. Matsuzawa, S. Matsuoka, and H. Waki. *Petascale general solver for semidefinite programming problems with over two million constraints*. In IEEE Int. Symp. Parallel and Distributed Processing (IPDPS 14) Proceedings, IEEE Computer Society, 2014.

[16] M. Girvan and M.E.J. Newman. *Community structure in social and biological networks*. Proc. Natl. Acad. Sci. USA, 99:7821–7826, 2002.

[17] T. Hoefler. *GreenGraph500 Submission Rules*, http://green.graph500.org/greengraph500rules.pdf.

[18] K. Iwabuchi, H. Sato, Y. Yasui, and K. Fujisawa. *Hybrid BFS approach using semi-external memory*. In Int. Workshop on High Performance Data Intensive Computing (HPDIC2014) Proceedings, 2014.

[19] K. Iwabuchi, H. Sato, Y. Yasui, K. Fujisawa, and S. Matsuoka. *NVM-based hybrid BFS with memory efficient data structure*. In IEEE Int. Conf. BigData 2014 Proceedings, IEEE Computer Society, 2014.

[20] H. Kwak, C. Lee, H. Park, and S. Moon. *What is Twitter, a social network or a news media?*. WWW'10 Proceedings of the 19th International Conference on World Wide Web, pp. 591–600, 2010

[21] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. *Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters*. Internet Mathematics, 6(1):29–123, 2009

[22] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. *Kronecker graphs: An approach to modeling networks*. J. Mach. Learning Res., 11:985–1042, 2010.

[23] J. Leskovec, D. Huttenlocher, and J. Kleinberg. *Signed networks in social media*. CHI'10 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1361–1370, 2010.

[24] J. Leskovec, D. Huttenlocher, and J. Kleinberg. *Predicting positive and negative links in online social networks*. WWW'10 Proceedings of the 19th International Conference on World Wide Web, pp. 641–650, 2010.

[25] J. McAuley and J. Leskovec. *Image labeling on a network: Using social-network metadata for image classification*. Computer Vision–ECCV 2012, Lecture Notes in Computer Science, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, C. Schmid, Eds. Springer International Publishing, 2012, Vol. 7575, pp. 828–841, 2012.

[26] U. Meyer and P. Sanders. *Δ-stepping: A parallelizable shortest path algorithm*. Journal of Algorithms, 49(1):114–152, October 2003.

[27] R.C. Murphy, K.B. Wheeler, B.W. Barrett, and J.A. Ang. *Introducing the Graph500*, Cray User Group 2010 Proceedings, 2010.

[28] R. Pearce, M. Gokhale, and N.M. Amato. *Multithreaded asynchronous graph traversal for in-memory and semi-external memory*. In High Performance Computing, Networking, Storage and Analysis (SC), (2010) International Conference Proceedings, pp. 1–11, Nov. 2010.

[29] F. Petrini, F. Checconi, J. Willcock, A. Lumsdaine, A.R. Choudhury, and Y Sabharval. *Breaking the speed and scalability barriers for graph exploration on distributed-memory machines*. In ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC12) Proceedings, IEEE Computer Society, 13, 2012.

[30] C. Seshadhri, A. Pinar, and T.G. Kolda. *An in-depth analysis of stochastic Kronecker graphs*. Journal of the ACM (JACM), 60(2):13, 2013.

[31] K. Ueno and T. Suzumura. *Highly scalable graph search for the Graph500 Benchmark*. In 21st Int. ACM Symp. High-Performance Parallel and Distributed Computing (HPDC'12) Proceedings, pp. 149–160, ACM, 2012.

[32] K. Ueno and T. Suzumura. *Parallel distributed breadth first search on GPU*. In IEEE Int. Conf. High Performance Computing (HiPC 2013) Proceedings, IEEE Computer Society, 2013.

[33] J. Yang and J. Leskovec. *Defining and evaluating network communities based on ground-truth*. MDS'12 Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics, Article No. 3, 2012.

[34] Y. Yasui, K. Fujisawa, K. Goto, N. Kamiyama, and M. Takamatsu. *NETAL: High-performance implementation of network analysis library considering computer memory hierarchy*. J. Oper. Res. Soc. Japan, 54(4):259–280, 2011.

[35] Y. Yasui, K. Fujisawa, and K. Goto. *NUMA-optimized parallel breadth-first search on multicore single-node system*. In IEEE Int. Conf. BigData 2013, IEEE Computer Society, 2013.

[36] Y. Yasui, K. Fujisawa, and Y. Sato. *Fast and energy-efficient breadth-first search on a single NUMA system*. Supercomputing, Lecture Notes in Computer Science, J.M. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer International Publishing, 2014, vol. 8488, pp. 365–381, 2014.

[37] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. *A scalable distributed parallel breadth-first search algorithm on BlueGene/L*. In ACM/IEEE Conf. Supercomputing (SC05) Proceedings, IEEE Computer Society, 25, 2005.