Fast and Energy-efficient Breadth-First Search on a Single NUMA System

Yuichiro Yasui[1], Katsuki Fujisawa[1], and Yukinori Sato[2]

[1] Kyushu University, Fukuoka, Japan
{y-yasui,fujisawa}@imi.kyushu-u.ac.jp
[2] JAIST, Ishikawa, Japan
yukinori@jaist.ac.jp

Abstract. Breadth-first search (BFS) is an important graph analysis kernel. The Graph500 benchmark measures a computer's BFS performance using the traversed edges per second (TEPS) ratio. Our previous nonuniform memory access (NUMA)-optimized BFS reduced memory accesses to remote RAM on a NUMA architecture system; its performance was 11 GTEPS (giga TEPS) on a 4-way Intel Xeon E5-4640 system. Herein, we investigated the computational complexity of the bottom-up, a major bottleneck in NUMA-optimized BFS. We clarify the relationship between vertex out-degree and bottom-up performance. In November 2013, our new implementation achieved a Graph500 benchmark performance of 37.66 GTEPS (fastest for a single node) on an SGI Altix UV1000 (one-rack) and 31.65 GTEPS (fastest for a single server) on a 4-way Intel Xeon E5-4650 system. Furthermore, we achieved the highest Green Graph500 performance of 153.17 MTEPS/W (mega TEPS per watt) on an Xperia-A SO-04E with a Qualcomm Snapdragon S4 Pro APQ8064.

Keywords: Breadth-first search, graph algorithms, parallel algorithms, multicore processing.

1 Introduction

The breadth-first search (BFS) is one of the most important graph analysis kernels. It can be used to obtain some properties of the connections between the nodes in a given graph. BFS can be used as not only a standalone kernel but also a subroutine in several applications such as connected components [2], maximum flow [1], centrality [3, 4], and clustering [5]. In particular, some recent streams have represented mathematical graphs over wide areas, and these kernels have been used for further analysis[1,2]. BFS has a linear theoretical complexity of $O(n+m)$ for a problem with $n = |V|$ vertices and $m = |E|$ edges in a given

[1] Stanford Network Analysis Project: http://snap.stanford.edu
[2] Human Brain Project: http://www.humanbrainproject.eu

graph $G = (V, E)$. This complexity is optimal for sequential computation; however, it is not efficient for parallel computation. Therefore, previous studies have proposed efficient parallel computation algorithms based on a *level-synchronized parallel BFS* for multicore single-node systems [12–16] and multicore multi-node systems [17–21]. Table 1 shows the BFS performance of each of these implementations in terms of the traversed edges per second (TEPS) ratio and TEPS ratio per watt. Our previous algorithm [16], which is based on Beamer's algorithm [15], achieves a performance of 11.2 GTEPS. Our algorithm was adapted to a nonuniform memory access (NUMA) architecture by efficiently managing computer resources, as a result of which its TEPS ratio was higher than that of other algorithms. In this study, we propose a fast and highly energy-efficient parallel BFS algorithm for a NUMA system that incorporates additional speedup techniques to achieve a performance of over 30 GTEPS, which is around three times faster than our previous algorithm. The primary contributions of this study are as follows:

1. An investigation of the major bottleneck in our NUMA-optimized BFS on a NUMA system.
2. A fast and highly energy-efficient BFS algorithm that applies degree-based speedup techniques on a NUMA system.

Table 1. BFS performance (TEPS ratio and TEPS/W) in related work

Authors	**Base architecture** (#CPU cores)	$\log_2 n$	m/n	**GTEPS**	**MTEPS/W**
Reference code [8]	4-way Intel Xeon E5-4640 (64)	27	16	0.1	0.20
Madduri et al. [12]	Cray MTA-2 (40)	21	512	0.5	-
Agarwal et al. [13]	4-way Intel Xeon 7500 (64)	20	64	1.3	-
Beamer et al. [14]	4-way Intel Xeon E7-8870 (80)	28	16	5.1	-
Yasui et al. [16]	4-way Intel Xeon E5-4640 (64)	26	16	11.2	17.39
This study	4-way Intel Xeon E5-4640 (64)	27	16	29.0	45.43
	4-way Intel Xeon E5-4650 (64)	27	16	31.7	41.01
	SGI Altix UV 1000 (512)	30	16	37.7	1.89
	Sony Xperia-A-SO-04E (4)	20	16	0.48	153.17
	ASUS Nexus 7 –2013 version– (4)	20	16	0.53	129.63

2 Brief Introduction to BFS and Related Work

2.1 BFS, Graph500, and Green Graph500

Breadth-First Search. 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 $s \in V$.

Graph500 and Green Graph500. The Graph500 benchmark[3] is designed to measure the computer performance for applications that require an irregular memory access pattern. Following its announcement in June 2010, the first Graph500 list was released in November 2010. This benchmark must perform the following steps:

1. **Generation.** This step generates the edge list of the Kronecker graph [10] using a recursive matrix (R-MAT) computation [11]. This generator requires the scale and the edgefactor as inputs. It outputs the 2^{scale} vertices and $2^{\text{scale}} \cdot \text{edgefactor}$ edges that 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 of the BFS.

Graph500 benchmark is based on the TEPS ratio, which is computed for a given graph and the BFS output [8]. The Green Graph500 benchmark[4] is designed to measure the energy efficiency of a computer by using the TEPS ratio per watt [9]. These lists have been updated biannually since their introduction.

2.2 CPU Affinity and Local Memory Allocation on NUMA System

We provide some definitions of processor affinity and memory policy for NUMA architecture processors. The mapping of CPU cores is represented by the processor ID, package ID, core ID, and SMT ID. The processor ID is a unique integer associated with the processing element, and it is used to bind threads to cores by invoking the system call `sched_setaffinity()`. The package ID is a unique integer associated with the physical chip, and it is used to bind data to the memory by invoking the system call `mbind()`. Finally, the core ID and the SMT ID are unique integers associated with the processing element in the physical chip and the processing thread in the processing element, respectively.

In NUMA architecture systems, the memory access time depends on the memory location relative to a processor. A processor can access its local memory faster than it can its remote (nonlocal) memory (i.e., memory local to another processor or memory shared between processors). In this study, we propose general techniques for processor affinity and memory binding for NUMA architecture systems. We have already applied similar techniques in our previous work [16] as well as in graph algorithms for the shortest path problem [6] and mathematical optimization problem [7]. We have implemented the ubiquity library for intelligently binding cores (ULIBC) [16] for an automatic configuration system for CPU affinity and memory binding on a NUMA architecture and Linux system, as shown in Fig. 1. Our library provides some APIs in the parallel region, such as `get_numa_procid()` and `get_numa_cputopo()`, which require the thread ID

[3] Graph500 benchmark: `http://www.graph500.org`
[4] Green Graph500 benchmark: `http://green.graph500.org`

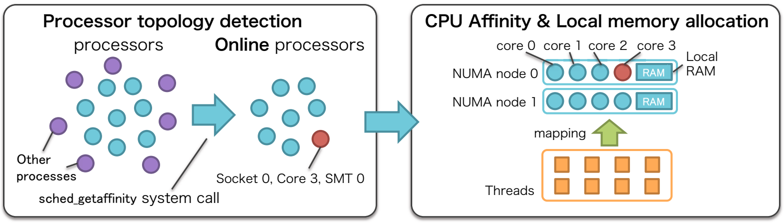


Fig. 1. Configuration of CPU affinity and local memory binding

and processor ID as inputs and return the processor ID for the Thread–CPU mapping and a tuple of (SMT ID, core ID, package ID) for the CPU topology, respectively.

2.3 NUMA-Optimized Hybrid BFS

Beamer et al. [14, 15] proposed a hybrid framework for the BFS algorithm (Algorithm 1) that reduced the number of edges explored. This algorithm combines two different traversal kernels: *top-down* and *bottom-up*. The former traverses *neighbors* Q^N of the *frontier* Q^F, whereas the latter finds the *frontier* from vertices in candidate neighbors (all unvisited vertices). This algorithm requires traversal (lines 6–9) and swaps Q^N and Q^F (line 10) at each level.

Algorithm 1. Hybrid BFS algorithm proposed by Beamer et al.

```
Input  : directed graph $G = (V, A^{\mathcal{F}}, A^{\mathcal{B}})$ and source vertex $s \in V$.
Data   : frontier queue $Q^F$, neighbor queue $Q^N$, and visited flag $visited$.
Output: predecessor map of BFS tree $\pi(v)$.
1  $\pi(v) \leftarrow -1, \forall v \in V$
2  $visited \leftarrow \{s\}$
3  $Q^F \leftarrow \{s\}$
4  $Q^N \leftarrow \emptyset$
5  while $Q^F \neq \emptyset$ do
6      if $is\_TopDown(Q^F, Q^N, visited)$ then
7          $Q^N \leftarrow$ Top-down$(G, Q^F, visited, \pi)$          /* Traversal */
8      else
9          $Q^N \leftarrow$ Bottom-up$(G, Q^F, visited, \pi)$        /* Traversal */
10     swap$(Q^F, Q^N)$                                               /* Swap */
```

Table 2 shows how a traversal policy is determined for *top-down* and *bottom-up* kernels (line 6). The traversal policy of the hybrid algorithm moves from *top-down* to *bottom-up* in the *growing* phase $|Q^F| < |Q^N|$, and it returns from *bottom-up* to *top-down* in the *shrinking* phase $|Q^F| \geq |Q^N|$. The algorithm uses the exact and approximate number of traversed edges $m_{\mathcal{F}}$ and $m'_{\mathcal{F}}$ in *top-down* and the approximate traversed edges of the *bottom-up*, $m'_{\mathcal{B}}$, which are defined as follows:

$$m_{\mathcal{F}} \leftarrow \left|\left\{(v,w) \in E \mid v \in Q^N,\ w \in A^{\mathcal{F}}(v)\right\}\right|, \tag{1}$$

$$m'_{\mathcal{F}} \leftarrow \left|Q^N\right| \cdot \text{edgefactor}, \tag{2}$$

$$m'_{\mathcal{B}} \leftarrow |V \setminus \mathit{visited}| \cdot \text{edgefactor} + |V|. \tag{3}$$

In addition, we determine the optimum values of the switching parameters α and β, which are set as 16 and 16 based on the actual execution time.

Table 2. Traversal policy selection

(a) Growing phase $|Q^F| < |Q^N|$

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	–	always

(b) Shrinking phase $|Q^F| \geq |Q^N|$

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		

Our NUMA-optimized algorithm, which is based on Beamer et al.'s hybrid algorithm, requires a given graph and working variables for a BFS to be divided into the local memory before the traversal. In our algorithm, the traversal phase avoids accessing the remote memory by using the following column-wise partitioning:

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

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

$$V_k = \left\{v_j \in V \mid j \in \left[\frac{kn}{\ell}, \frac{k(n+1)}{\ell}\right)\right\}, \tag{5}$$

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

$$A_k^{\mathcal{F}}(v) = \left\{w \in \left\{V_k \cap A(v)\right\}\right\}, v \in V, \qquad A_k^{\mathcal{B}}(w) = \left\{v \in A(w)\right\}, w \in V_k. \tag{6}$$

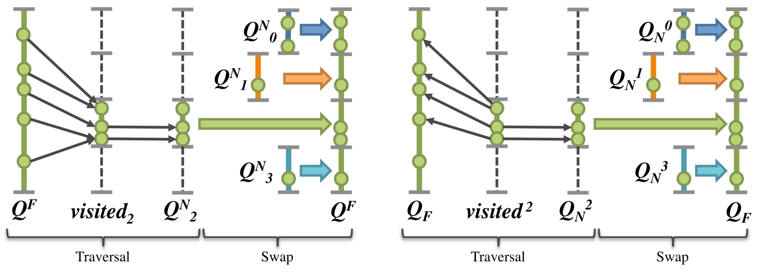
Furthermore, the working spaces Q_k^N, visited$_k$, and π_k for partial vertices V_k are allocated to the local memory on k-th NUMA node with memory pinned. Algorithms 2 and 3 and Figs 2(a) and 2(b) describe the *top-down* and *bottom-up* kernels for the NUMA-optimized BFS algorithm. Both algorithms bind the threads T_k to processors and local memory at k-th NUMA node without accessing the remote memory. Their respective computational complexities are $O(m)$ and $O(m \cdot \text{diam}_G)$, where m is the number of edges and diam$_G$ is the diameter of the given graph; this is the same complexity as that of Beamer et al.'s previous hybrid algorithm. Therefore, the hybrid algorithm that combines these algorithms has $O(m \cdot \text{diam}_G)$ complexity. However, the actual CPU time of the hybrid algorithm is shorter than that of top-down only for a small-world network such as a Kronecker graph.

Algorithm 2. NUMA-optimized top-down BFS.

```
Input  : number of CPU sockets $\ell$, NUMA node IDs
         $k = \{0, 1, \cdots, \ell - 1\}$, directed graph $G = \{G_k\} = \{(V_k, A_k^{\mathcal{F}})\}$,
         frontier queue (local copy) $Q^F$, visited flag $visited = \{visited_k\}$,
         and predecessor map of BFS tree $\pi = \{\pi_k\}$.
Output: neighbor queue $Q^N = \{Q_k^N\}$.
1  $T_k \leftarrow \emptyset, \forall k \in \ell$
2  fork()
3  $t \leftarrow$ omp_get_thread_num()
4  sched_cpuaffinity(get_numa_procid($t$))
5  $(i, j, k) \leftarrow$ get_numa_cputopo(get_numa_procid($t$))
6  $T_k \leftarrow T_k \cup \{(i, j)\}$
7  $Q_k^N \leftarrow \emptyset$
8  for $v \in Q^F$ in parallel($T_k$) do
9      for $w \in A_k^{\mathcal{F}}(v)$ do
10         if $w \notin visited_k$ atomic then
11             $\pi_k(w) \leftarrow v$
12             $visited_k \leftarrow visited_k \cup \{w\}$
13             $Q_k^N \leftarrow Q_k^N \cup \{w\}$
14 join()
```

Algorithm 3. NUMA-optimized bottom-up BFS.

```
Input  : number of CPU sockets $\ell$, NUMA node IDs
         $k = \{0, 1, \cdots, \ell - 1\}$, directed graph $G = \{G_k\} = \{(V_k, A_k^{\mathcal{F}})\}$,
         frontier queue (local copy) $Q^F$, visited flag $visited = \{visited_k\}$,
         and predecessor map of BFS tree $\pi = \{\pi_k\}$.
Output: $Q^N = \{Q_k^N\}$ : neighbor queue.
1  $T_k \leftarrow \emptyset, \forall k \in \ell$
2  fork()
3  $t \leftarrow$ omp_get_thread_num()
4  sched_cpuaffinity(get_numa_procid($t$))
5  $(i, j, k) \leftarrow$ get_numa_cputopo(get_numa_procid($t$))
6  $T_k \leftarrow T_k \cup \{(i, j)\}$
7  $Q_k^N \leftarrow \emptyset$
8  for $w \in V_k \setminus visited_k$ in parallel($T_k$) do
9      for $v \in A_k^{\mathcal{B}}(w)$ do
10         if $v \in Q^F$ then
11             $\pi_k(w) \leftarrow v$
12             $visited_k \leftarrow visited_k \cup \{w\}$
13             $Q_k^N \leftarrow Q_k^N \cup \{w\}$
14             break
15 join()
```



(a) Top-down on 2-th NUMA node. (b) Bottom-up on 2-th NUMA node.

Fig. 2. Overview of NUMA-optimized BFS on ℓ-way NUMA system ($\ell = 4$)

3 Degree-Aware BFS

3.1 Bottleneck Analysis of Hybrid-BFS

In this section, we explain our proposed *Degree-aware BFS* algorithm that improves the major bottleneck in NUMA-optimized BFS. First, we show the major bottleneck in NUMA-optimized BFS, namely, the bottom-up kernel. The bottom-up step checks that each unvisited vertex has an adjacent vertex that connects vertices in the frontier. Therefore, this step reduces unnecessary edge traversals if an adjacency vertex that has already been visited with high probability is allocated at a higher position of each adjacency vertex list. It is difficult to obtain the optimal ordering for the adjacency vertex list. however, we focus on the out-degree $\deg_G(v)$ of each vertex $v \in V$, which is defined as follows:

$$\deg_G(v) = |A(v)|, v \in V. \tag{7}$$

Table 3 shows a comparison of the number of traversed edges for each level in *top-down* and *bottom-up*, and each ordering, such as *Ascending*, *Randomized*, and *Descending*. *Ascending* and *Descending* construct an adjacency vertex list $A(v)$ that is sorted by $\deg_G(w), w \in A(v)$ in ascending and Descending order, respectively. *Randomized* constructs an adjacency vertex list $A(v)$ that is randomized. The table shows that most traversed edges were concentrated in Level-2 and that the number of traversed edges is affected by each ordering.

Fig. 3 shows the distribution of the loop count τ of the bottom-up step in each level. These figures can be used to easily understand the properties of first adjacency lists. The maximum count of the bottom-up loop ($\max \tau = 28$) using descending order is much smaller than that ($\max \tau = 5,873$) using ascending order. This clearly suggests that the ordering strategies affect the loop count.

Finally, we investigate the breakdown of processing in each vertex for each adjacency ordering shown in Table 4. The table shows that the number of zero-degree vertices is half the total number of vertices. These orderings have the same property in that most traversals of unvisited vertices occur in bottom-up searching. In particular, when using descending order, a BFS requires the most total CPU time in bottom-up at the first loop ($\tau = 1$).

Table 3. Number of traversed edges in a BFS for Kronecker graph with scale 27

			Hybrid algorithm			
Level	**Top-down**	**Bottom-up**	**Step**	**Ascending**	**Randomized**	**Descending**
0	22	4,223,250,243	T	22	22	22
1	239,930	3,258,645,723	T	239,930	239,930	239,930
2	1,040,268,126	83,878,899	**B**	**848,743,124**	**150,006,673**	**83,878,899**
3	3,145,608,885	19,616,130	B	19,935,737	19,742,764	19,616,130
4	37,007,608	139,606	B	139,868	139,817	139,606
5	98,339	41,846	B	41,846	41,846	41,846
6	260	41,586	T	260	260	260
Total	4,223,223,170	7,585,614,033	–	869,100,787	170,171,312	103,916,693
%	100 %	179.6 %		20.6 %	4.0 %	2.5 %

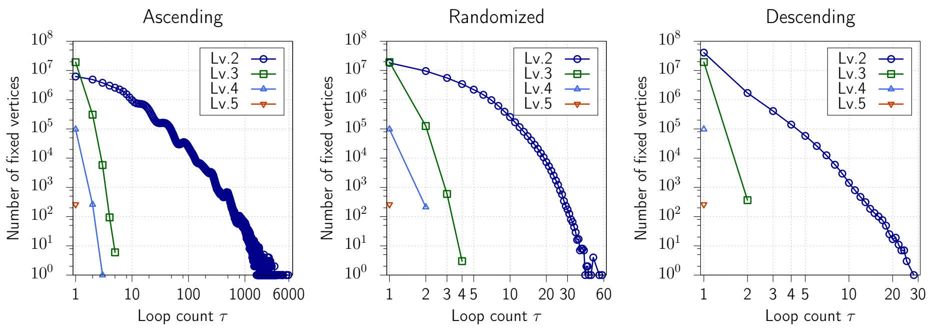


Fig. 3. Distribution of the loop count τ of the bottom-up step at each level in a BFS for Kronecker graph with SCALE27 and edgefactor 16

Table 4. Breakdown of bottleneck in a BFS of a Kronecker graph with scale 27

(a) Ascending order

Component	0-degree	Bottom-up ($\tau = 1$)	Bottom-up ($\tau \geq 2$)	Top-down
#vertices	71,140,085	**25,489,401**	37,331,644	215,070
Ratio	53.00 %	**18.99 %**	27.81 %	0.16 %

(b) Randomized order

Component	0-degree	Bottom-up ($\tau = 1$)	Bottom-up ($\tau \geq 2$)	Top-down
#vertices	71,140,085	**37,663,087**	25,157,958	215,070
Ratio	53.00 %	**28.06 %**	18.74 %	0.16 %

(c) Descending order

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

3.2 Degree-Aware BFS on Degree-Aware Graph Representation

The reference to the adjacent vertices require many in-directing accesses via an index array and an adjacency edge array of the compressed sparse row (CSR) format graph. However, we clarified that the major bottleneck of the hybrid BFS algorithm is the edge traversal of the first adjacent vertex in the bottom-up step in the previous subsection. Then, we separated the standard CSR graph into the *highest-degree adjacency vertex list* $A^{\mathcal{B}+}$ and resting CSR graph $A^{\mathcal{B}-}$. The highest-degree adjacency vertex $A^{\mathcal{B}+}(v)$ for each vertex v contains an adjacency vertex w, whose maximum degree is given as follows:

$$A^{\mathcal{B}+}(v) = \mathop{\arg\max}_{w \in A^{\mathcal{B}}(v)} \{|\mathrm{d}_G(w)|\}, v \in V. \tag{8}$$

This graph representation requires additional computational overhead only for sorting the adjacency vertex list, and it does not suffer from the problem of requiring increasing memory. We focus on the fact that half of the total number of vertices are zero-degree vertices. This property does not have much effect on 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 of zero-degree vertices, we propose *zero-degree vertex suppression*, which renumbers the vertex ID of only each nonzero vertice during graph construction. Algorithm 4 describes the degree-aware bottom-up BFS. It uses a graph representation that separates the highest-degree adjacency vertex $A^{\mathcal{B}+}$ and the resting adjacency vertices $A^{\mathcal{B}-}$. This algorithm separates two major loops for this purpose, such as lines 8–13 and lines 14–20.

We compared our speedup techniques—“highest-degree adjacency vertex list (high-deg)” and “zero-degree suppression (zero-deg)”—for a Kronecker graph with scale 27 on a SandyBridge-EP system, the results of which are shown in Table 5. Zero-deg showed two times improved performance relative to our previous NUMA-optimized BFS. In comparison, high-deg showed little effect (a speedup of about 34%) on the BFS performance. However, a combination of these two methods showed 2.68 times faster performance.

Table 5. Comparison of our speedup techniques on Sandybridge-EP

Implementation	SCALE	GTEPS	Speedup
NUMA-opt. [16]	27	10.85	× 1.00
NUMA-opt. + High-deg	27	14.55	× 1.34
NUMA-opt. + Zero-deg	27	22.01	× 2.03
NUMA-opt. + High-deg + Zero-deg	27	29.03	× 2.68

Table 6. Machine environments

Machine	Processor	$p = \ell \times t$	RAM	LLC	CC
Westmere-EP	Intel Xeon E7-4870 2.93GHz	24 = 2× 12	96 GB	12 MB	gcc-4.4
Magny-Cours	AMD Opteron 6174 2.20GHz	48 = 8× 6	256 GB	512 KB	gcc-4.8
Westmere-EX	Intel Xeon E7-4870 2.40GHz	80 = 4× 20	512 GB	30 MB	gcc-4.4
SandyBridge-EP	Intel Xeon E5-4640 2.40GHz	64 = 4× 16	512 GB	20 MB	gcc-4.4
SandyBridge-EP (2.7GHz)	Intel Xeon E5-4650 2.70GHz	64 = 4× 16	512 GB	20 MB	gcc-4.4
Altix UV1000	Intel Xeon E7-8837 2.67GHz	512 = 64× 8	4096 GB	24 MB	icc-12.1

Algorithm 4. Degree-aware NUMA-optimized bottom-up BFS.

```
Input  : number of CPU sockets $\ell$, NUMA node IDs $k = \{0, 1, \cdots, \ell - 1\}$,
         directed graph $G = \{G_k\} = \{(V_k, A_k^{\mathcal{B}+}, A^{\mathcal{B}-})\}$, frontier queue
         (local copy) $Q^F$, visited flag $visited = \{visited_k\}$, and
         predecessor map of BFS tree $\pi = \{\pi_k\}$.
Output: neighbor queue $Q^N = \{Q_k^N\}$.
1  $T_k \leftarrow \emptyset, \forall k \in \ell$
2  fork()
3  $t \leftarrow$ omp_get_thread_num()
4  sched_cpuaffinity(get_numa_procid($t$))
5  $(i, j, k) \leftarrow$ get_numa_cputopo(get_numa_procid($t$))
6  $T_k \leftarrow T_k \cup \{(i, j)\}$
7  $Q_k^N \leftarrow \emptyset$
8  for $w \in V_k \setminus visited_k$ parallel($T_k$) do
9      $v \leftarrow A_k^{\mathcal{B}+}(w)$
10     if $v \in Q^F$ then
11         $\pi_k(w) \leftarrow v$
12         $visited_k \leftarrow visited_k \cup \{w\}$
13         $Q_k^N \leftarrow Q_k^N \cup \{w\}$
14 for $w \in V_k \setminus visited_k$ parallel($T_k$) do
15     for $v \in A_k^{\mathcal{B}-}(w)$ do
16         if $v \in Q^F$ then
17             $\pi_k(w) \leftarrow v$
18             $visited_k \leftarrow visited_k \cup \{w\}$
19             $Q_k^N \leftarrow Q_k^N \cup \{w\}$
20             break
21 join()
```

4 Numerical Results

4.1 Implementation and Machine Environments

Table 6 shows the environments of each NUMA architecture system, such as the processor, RAM size, last level cache (LLC) size, and C compiler (CC). Each system has a maximum of p threads on ℓ-way NUMA nodes, each of which contains t logical cores. We used the optimization option setting of `-O2`.

4.2 BFS Performance on Graph500 Benchmark

Performance Variation with Problem Size. Table 7 shows a comparison of the performance of the reference code (Graph500 version 2.1.4), our previous algorithm (NUMA-optimized), and our algorithm (Degree-aware, This study) for a Kronecker graph with edgefactor 16 on the SandyBridge-EP. The speedup ratio of our algorithm with respect to the reference code generally increases with scale for the Kronecker graph. Each blank in the table indicates that the reference code generated a serious error and was aborted. Fig. 4 shows a comparison of the performance of our BFS on different NUMA systems with respect to scale. However, our BFS causes performance degradation because the frontier queue size is larger than the last level cache size, as also reported in Agarwal et al. [13].

Table 7. Performance (GTEPS and speedup ratio) of the reference code and our algorithm (NUMA-optimized and Degree-aware) on SandyBridge-EP

	Reference	**NUMA-optimized** [16]		**Degree-aware**	
SCALE	*GTEPS*	*GTEPS*	*Speedup*	*GTEPS*	*Speedup*
20	0.358	4.578	× 12.8	4.994	× 14.0
21	0.233	6.317	× 27.1	8.649	× 37.1
22	0.101	8.433	× 83.7	13.896	×137.6
23	0.129	10.023	× 77.6	17.859	×138.4
24	0.123	10.829	× 88.3	22.317	×181.4
25	0.107	11.155	×104.4	25.269	×236.2
26	0.097	11.149	×114.9	26.675	×275.0
27	0.087	10.854	×124.5	29.034	×333.7
28	–	9.887	–	23.522	–
29	–	9.393	–	21.381	–

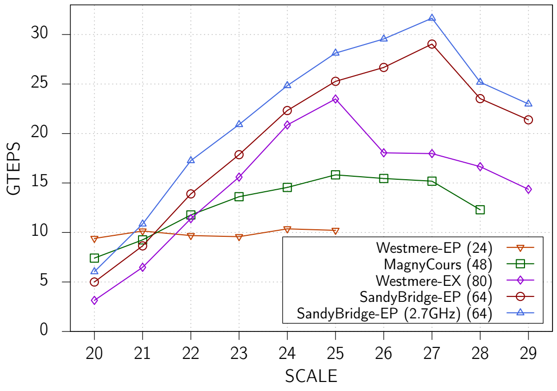
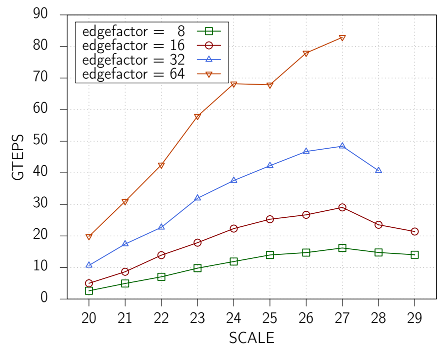
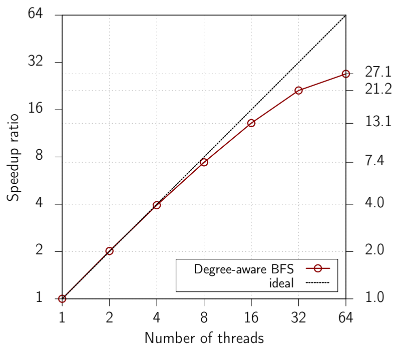


Fig. 4. Variation in BFS performance (GTEPS) with problem size on NUMA machines

Performance Variation with Edgefactor and Scalability. Fig. 5(a) shows how the performance of our algorithm varied with the scale and edgefactor of the Kronecker graph. The blanks in the figures indicate points at which the system had insufficient memory space to execute the algorithm. For the Kronecker graph with a scale 27, our BFS attained a peak TEPS for edgefactors of 8, 16, 32, and 64. Our BFS is more efficient for a Kronecker graph with a large edgefactor (64) than for one with a small edgefactor (8). This speedup afforded by *bottom-up* reduces the edge traversals required for a dense graph (large edgefactor). Furthermore, fig. 5(b) shows that our BFS with 64 threads is 27.1 times faster than sequencial.



(a) Performance (GTEPS) varied with problem size and edgefactor



(b) Strong scaling for SCALE 27 and edgefactor 16

Fig. 5. Performance (GTEPS) and scalability of our algorithm on SandyBridge-EP

4.3 BFS Performance on SGI Altix UV1000 System

We discuss the BFS performance on an SGI Altix UV1000, which actualizes massive thread parallel computing based on cache-coherent nonuniform memory access (ccNUMA). A rack of SGI Altix UV1000 systems contains 192 NUMA nodes with an Intel Xeon CPU E7-8837 2.67 GHz and 64 GB RAM and generates up to 8 threads in parallel with hyperthreading disabled. Our BFS requires edge traversal to be executed in local RAM (Algorithm 1, lines 6–9) and the frontier queue and the neighbor queue for the next level to be swapped in remote RAM (Algorithm 1, line 10). We do not consider the swap operation as a bottleneck because most CPU time is spent on the traversal operations on other NUMA systems with a few CPU sockets. Table 8 shows the CPU time and the variation of the ratio of traversals and swaps with the number of threads p set as 128 (one-fourth of a rack), 256 (one-half of a rack), and 512 (one rack) for Kronecker graph with scale 30. As the number of threads increases, the bottleneck moves from the traversal operation in local memory to the swap operation in remote memory. Therefore, we focus on the fast computation of the swap operation.

Table 8. TEPS, CPU time, and ratio of traversal and swap on SGI Altix UV1000

p	GTEPS	Traversal on Local RAM	Swap on Remote RAM
128	18.76	732.80 ms (80%)	182.10 ms (20%)
256	26.17	437.90 ms (67%)	218.00 ms (33%)
512	37.70	257.90 ms (57%)	196.60 ms (43%)

4.4 BFS Performance on Real-World Networks

We verify our BFS performance by using real-world networks on a Sandybridge-EP system. Table 9 shows the graph sizes, graph properties, and GTEPS using our BFS for each network instance. Here, diam'_G indicates the approximation of the diameter on each network using the maximum hops for each vertex of 64 BFS iterations. USA-road-d and wiki-Talk have approximately the same edgefactor, but the latter shows 4–11 times faster than the former owing to its diameter being a thousand times smaller relatively. On the other hand, LiveJournal and twitter have approximately the same diameter, but the latter shows 3–5 times faster than the former owing to its edgefactor being 1.68 times larger relatively. In addition, twitter and friendster show similar BFS performances of approximately 10 GTEPS because they have similar edgefactor and similar diameters. Therefore, we verify whether our BFS is affected by using both the edgefactor and diameter of the network. From these numerical results, we could achieve high performance for large-scale small-world networks with a large edgefactor.

Table 9. BFS performance of real-world network on Sandybridge-EP system

	Graph size		**edgefactor**	**Diameter**	**GTEPS**				
Instance	n	m	m/n	diam'_G	min	1/4	median	3/4	max
wiki-Talk [23, 24]	2.39 M	5.02 M	2.1	8	0.29	0.61	0.75	0.87	1.26
USA-road-d [25]	23.95 M	58.33 M	2.4	8,098	0.07	0.08	0.09	0.09	0.11
LiveJournal [26, 27]	4.85 M	68.99 M	14.2	16	2.76	3.76	4.07	4.32	4.94
twitter [28]	61.58 M	1,468.37 M	23.8	16	7.58	10.02	10.90	12.68	24.09
friendster [29]	65.61 M	1,806.07 M	27.5	25	4.89	9.61	10.74	11.29	11.81

5 Energy Efficiency of Our BFS

Thus far, we have discussed the BFS performance in terms of only the speed. This section discusses the BFS performance of our implementation in terms of energy efficiency. Our speedup techiques are aimed at not only fast computation but also energy efficiency, as described in [4].

Fig. 6 shows the performance (GTEPS) and energy-efficiency (MTEPS/W) of our Degree-aware BFS, our NUMA-optimized BFS, and the Graph500 reference code for each CPU affinity on the SandyBridge-EP. If the number of threads is the same, our Degree-aware BFS achieves a high TEPS and a high TEPS/W with a larger number of sockets. On the other hand, if the number of sockets is the same, our BFS achieves a high TEPS and a high TEPS/W with a larger number of threads.

As an energy-efficient machine environment, we use Android-based devices, which have highly energy-efficient ARM or Snapdragon processors. We develop a native binary code based on the C/C++ programming language by using the Android native development kit (NDK). The Android NDK supports multithreaded computation using the OpenMP library and the atomic functions of the GCC extension. It enables binary code to be transferred to the device and then be execute using Android developer tools. This developer environment was published on the Android Developers website[3].

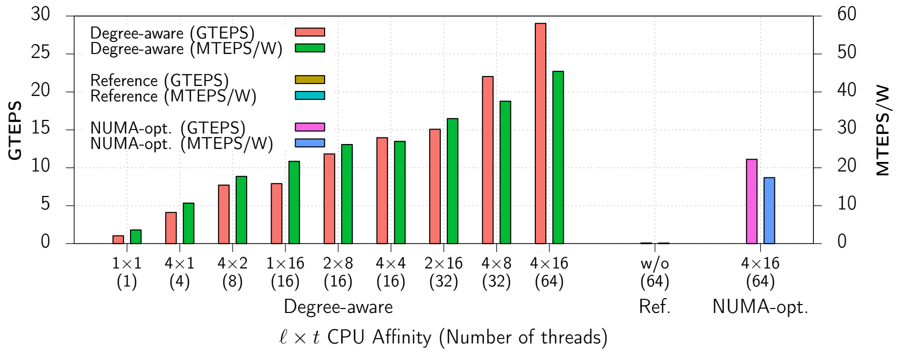


Fig. 6. Performance (GTEPS) and Energy-efficiency (MTEPS/W) of our Degree-aware BFS, our NUMA-optimized BFS, and Graph500 reference code for Kronecker graph with SCALE 27 on SandyBridge-EP with ℓ-NUMA-nodes × t-threads CPU-affinity

Fig. 7 shows the performance of our degree-aware BFS and the reference BFS on Sony's Android-based XperiaA SO-04E smartphone, which has a Qualcomm Snapdragon S4 Pro APQ8064 1.5 GHz processor and 2 GB RAM. Our BFS is assumed to be executed on a NUMA architecture system. It is difficult to achieve high performance directly using NUMA-optimized speedup techniques on a single CPU. However, we designed this algorithm to reduce unnecessary memory accesses and overheads of atomic operations, and as a result, we could achieve high performance even on Android devices, which do not have a NUMA architecture system. This figure shows that the reference code causes performance degradation. In comparison, our BFS achieves a maximum performance of 477.63 MTEPS for a Kronecker graph with scale 20 on XperiaA SO-04E with 4 threads.

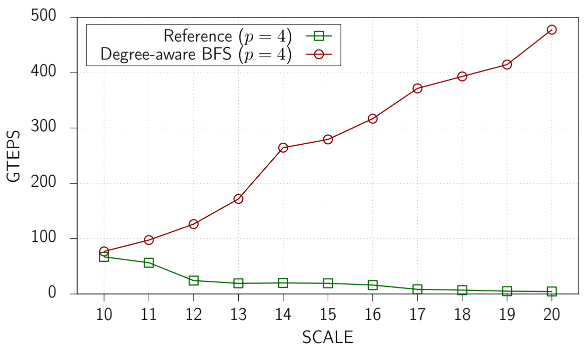


Fig. 7. MTEPS of reference BFS and Degree-aware BFS on XperiaA SO-04E

Table 10 summaries the energy efficiency in terms of MTEPS, effective power (W), and MTEPS/W for BFS on a Kronecker graph with scale 20. The table shows that all algorithms have an effective power of around 3.0 W for BFS execution, suggesting that the effective power is not strongly affected by the number of threads and the algorithm used. With regard to energy-efficient computation, our degree-aware BFS is around 100 times faster than the reference code for roughly the same effective power of 3.0 W; specifically, our BFS shows an energy-efficient performance of 153.17 MTESP/W.

Table 10. Energy efficiency of BFS for Kronecker graph with on XperiaA SO-04E

Implementation	SCALE	MTEPS	watt	MTEPS/W
Reference ($p = 1$)	20	3.25	3.15	1.03
Reference ($p = 4$)	20	4.58	3.22	1.42
Degree-aware ($p = 1$)	20	136.29	3.23	42.25
Degree-aware ($p = 2$)	20	248.08	2.99	82.92
Degree-aware ($p = 4$)	20	477.63	3.12	153.17

6 Conclusion

In this study, we investigate the major bottleneck in our previous NUMA-optimized BFS algorithm and apply degree-aware speedup techniques to it. Our new implementation achieved a BFS search performance of 31.65 GTEPS on 37.66 GTEPS on an SGI Altix UV1000 and a 4-way Intel Xeon E5-4650 system, which were ranked as the 50th (fastest on single node) and 51st (fastest on single server) best performances on the Graph500 list in November 2013, respectively. In addition, our BFS performance on a Sony Xperia-A-SO-04E and ASUS Nexus 7 (2013 version) was ranked as the first and second best performances in terms of energy efficiency on the Green Graph500 list in November 2013, respectively. Finally, further studies will be required to clarify the relationship between the theoretical complexity of our BFS and the small-world property.

Acknowledgments. We appreciate the valuable comments and suggestions from the anonymous reviewers. This research was supported by the Core Research for Evolutional Science and Technology (CREST) program of the Japan Science and Technology Agency (JST) and by the JAIST Research Center for Advanced Computing Infrastructure.

References

1. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM 19(2), 248–264 (1972)
2. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. MIT Press, Cambridge (1990)
3. Brandes, U.: A Faster Algorithm for Betweenness Centrality. J. Math. Sociol. 25(2), 163–177 (2001)

4. Frasca, M., Madduri, K., Raghavan, P.: NUMA-Aware Graph Mining Techniques for Performance and Energy Efficiency. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC 2012), pp. 1–11. IEEE Computer Society (2012)
5. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proc. Natl. Acad. Sci. USA 99, 7821–7826 (2002)
6. Yasui, Y., Fujisawa, K., Goto, K., Kamiyama, N., Takamatsu, M.: NETAL: High-performance Implementation of Network Analysis Library Considering Computer Memory Hierarchy. J. Oper. Res. Soc. Japan 54(4), 259–280 (2011)
7. Fujisawa, K., Endo, T., Yasui, Y., Sato, H., Matsuzawa, N., Matsuoka, S., Waki, H.: Petascale General Solver for Semidefinite Programming Problems with Over Two Million Constraints. In: Proc. IEEE Int. Symp. Parallel and Distributed Processing (IPDPS 2014). IEEE Computer Society (2014)
8. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the Graph500. In: Cray User Group 2010 Proceedings (2010)
9. Hoefler, T.: GreenGraph500 Submission Rules, `http://green.graph500.org/greengraph500rules.pdf`
10. Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., Ghahramani, Z.: Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learning Res. 11, 985–1042 (2010)
11. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A Recursive Model for Graph Mining. In: Proc. 4th SIAM Int. Conf. Data Mining, pp. 442–446. SIAM (2004)
12. Bader, D.A., Madduri, K.: Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In: Proc. 2006 Int. Conf. Parallel Processing (ICPP 2006), pp. 523–530. IEEE Computer Society (2006)
13. Agarwal, V., Petrini, F., Pasetto, D., Bader, D.A.: Scalable Graph Exploration on Multicore Processors. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC 2010), pp. 1–11. IEEE Computer Society (2010)
14. Beamer, S., Asanović, K., Patterson, D.A.: Searching for a Parent Instead of Fighting Over Children: A Fast Breadth-first Search Implementation for Graph500. EECS Department, University of California, UCB/EECS-2011-117, Berkeley, CA (2011)
15. Beamer, S., Asanović, K., Patterson, D.A.: Direction-optimizing Breadth-first Search. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC 2012), p. 12. IEEE Computer Society (2012)
16. Yasui, Y., Fujisawa, K., Goto, K.: NUMA-optimized Parallel Breadth-first Search on Multicore Single-node System. In: Proc. IEEE Int. Conf. BigData 2013. IEEE Computer Society (2013)
17. Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., Catalyurek, U.: A Scalable Distributed Parallel Breadth-first Search Algorithm on BlueGene/L. In: Proc. ACM/IEEE Conf. Supercomputing (SC 2005), p. 25. IEEE Computer Society (2005)
18. Buluç, A., Madduri, K.: Parallel Breadth-first Search on Distributed Memory Systems. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC11), p. 65. ACM (2011)
19. Petrini, F., Checconi, F., Willcock, J., Lumsdaine, A., Choudhury, A.R., Sabharval, Y.: Breaking the Speed and Scalability Barriers for Graph Exploration on Distributed-memory Machines. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC 2012), p. 13. IEEE Computer Society (2012)

20. Ueno, K., Suzumura, T.: Highly Scalable Graph Search for the Graph500 Benchmark. In: Proc. 21st Int. ACM Symp. High-Performance Parallel and Distributed Computing (HPDC 2012), pp. 149–160. ACM (2012)
21. Ueno, K., Suzumura, T.: Parallel Distributed Breadth First Search on GPU. In: Proc. IEEE Int. Conf. High Performance Computing (HiPC 2013). IEEE Computer Society (2013)
22. McAuley, J., Leskovec, J.: Image Labeling on a Network: Using Social-Network Metadata for Image Classification. In: Fitzgibbon, A., Lazebnik, S., Perona, P., Sato, Y., Schmid, C. (eds.) ECCV 2012, Part IV. LNCS, vol. 7575, pp. 828–841. Springer, Heidelberg (2012)
23. Leskovec, J., Huttenlocher, D., Kleinberg, J.: Signed Networks in Social Media. In: CHI (2010)
24. Leskovec, J., Huttenlocher, D., Kleinberg, J.: Predicting Positive and Negative Links in Online Social Networks. In: WWW (2010)
25. The 9th DIMACS Implementation Challenge, `http://www.dis.uniroma1.it/~challenge9/`
26. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group Formation in Large Social Networks: Membership, Growth, and Evolution. In: KDD (2006)
27. Leskovec, J., Lang, K., Dasgupta, A., Mahoney, M.: Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics 6(1), 29–123 (2009)
28. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a Social Network or a News Media? In: Proceedings of the 19th International Conference on World Wide Web (WWW 2010), pp. 591–600 (2010)
29. Yang, J., Leskovec, J.: Defining and Evaluating Network Communities based on Ground-truth. In: ICDM (2012)