NUMA-optimized Parallel Breadth-first Search on Multicore Single-node System

Yuichiro Yasui
Chuo University
Tokyo, Japan
yasui@indsys.chuo-u.ac.jp

Katsuki Fujisawa
Chuo University
Tokyo, Japan
fujisawa@indsys.chuo-u.ac.jp

Kazushige Goto
Intel Corporation
Hillsboro, OR, USA
kazushige.goto@intel.com

***Abstract*—The breadth-first search (BFS) is one of the most important kernels in graph theory. The Graph500 benchmark measures the performance of any supercomputer performing a BFS in terms of traversed edges per second (TEPS). Previous studies have proposed hybrid approaches that combine a well-known *top-down* algorithm and an efficient *bottom-up* algorithm for large frontiers. This reduces some unnecessary searching of outgoing edges in the BFS traversal of a small-world graph, such as a Kronecker graph. In this paper, we describe a highly efficient BFS using column-wise partitioning of the adjacency list while carefully considering the non-uniform memory access (NUMA) architecture. We explicitly manage the way in which each working thread accesses a partial adjacency list in local memory during BFS traversal. Our implementation has achieved a processing rate of 11.15 billion edges per second on a 4-way Intel Xeon E5-4640 system for a scale-26 problem of a Kronecker graph with 2^{26} vertices and 2^{30} edges. Not all of the speedup techniques in this paper are limited to the NUMA architecture system. With our winning Green Graph500 submission of June 2013, we achieved 64.12 GTEPS per kilowatt hour on an ASUS Pad TF700T with an NVIDIA Tegra 3 mobile processor.**

***Keywords*-Breadth-first search, graph algorithms, parallel algorithms, multicore processing**

I. Introduction

The breadth-first search (BFS) is one of the most important graph kernels. It obtains some properties of the connections between the nodes in a given graph. The BFS is not only utilized as a standalone kernel, but also as a subroutine in several applications, such as connected components, maximum flow, centrality [1], and clustering. In particular, some recent streams have represented mathematical graphs over wide areas, and these kernels have been used for further analysis[1,2]. The BFS has a linear theoretical complexity of $O(m)$ for a problem with n vertices and m edges in a given graph G. This complexity is optimal for sequential computation, but is not efficient for parallel computation. Therefore, previous studies have proposed efficient algorithms based on a *level-synchronized parallel BFS* in parallel computation on multicore single-node systems [6], [7], [8], [9] and multicore multi-node systems [10], [11], [12], [13]. Table I gives the performance of the BFS for each of these implementations in terms of the traversed edges per second (TEPS) ratio. The algorithm of Beamer et al. [8] achieves 5.1 GTEPS on a CPU-based single node, giving it a higher ratio than that of the other algorithms.

Table I
BFS PERFORMANCE (TEPS RATIO) IN RELATED WORK

Authors	Base architecture	Graph size	TEPS
Madduri et al. [6]	Cray MTA-2 (40 procs)	$n = 2^{21}$, $m = 2^{30}$	0.5 G
Agarwal et al. [7]	4-way Intel Xeon 7500	$n = 2^{20}$, $m = 2^{24}$	1.3 G
Beamer et al. [8]	4-way Intel Xeon E7-8870	$n = 2^{28}$, $m = 2^{32}$	5.1 G
This paper	4-way Intel Xeon E5-4640	$n = 2^{26}$, $m = 2^{30}$	11.15 G

In this paper, we explain a highly efficient parallel BFS algorithm on a non-uniform memory access (NUMA) architecture. Our implementation is based on the hybrid algorithm of Beamer et al., but it has been adapted to the NUMA architecture by efficiently managing the computer resources, allowing it to achieve 11.15 GTEPS. The primary contributions of this paper are the following:

1) An investigation of the system topology on multiple NUMA architecture processors, with automated configuration of CPU affinity and memory binding.
2) A highly efficient BFS algorithm based on column-wise partitioning for the adjacency list on a NUMA architecture system.

II. Graph500

The Graph500 benchmark[3] is designed to measure the performance of a computer system for applications that require an irregular memory access pattern. Following its announcement in June 2010, the Graph500 list was released in November 2010. The list has been updated biannually ever since. Reference implementations of the benchmark are provided for several platforms, including sequential platforms, OpenMP, the Cray XMT, and MPI. An implementation of this benchmark must perform the following steps:

1) Generation (untimed).: This step generates the edge list of the Kronecker graph [4] using a recursive matrix (R-MAT) computation [5].

2) Construction (timed).: This step constructs the graph representation from the edge list in Step 1.

[1]Stanford Network Analysis Project: `http://snap.stanford.edu`.
[2]Human Brain Project: `http://www.humanbrainproject.eu`.
[3]Graph500 benchmark: `http://www.graph500.org`.

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

3) BFS iterations (timed).: This step iterates the timed *BFS*-phase and the untimed *verify*-phase 64 times. The *BFS*-phase executes the BFS for each source, and the *verify*-phase confirms the output of the BFS.

This benchmark is based on the TEPS ratio, which is computed for a given graph and the BFS output. Submission against this benchmark must report five TEPS ratios: the minimum, first quartile, median, third quartile, and maximum.

III. Parallel BFS Algorithm

A. Level-synchronized Parallel BFS

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 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, it represents a tree with the root vertex s. However, some applications, such as the *betweenness centrality* [1], require all of the parents for each vertex, which is equivalent to the number of hops from the source. Therefore, the output predecessor map is represented as a directed adjacency graph (DAG). In this paper, we focus on the Graph500 benchmark, and assume that the BFS output is a predecessor map that is represented by a tree.

Algorithm 1 is a fundamental parallel algorithm for a BFS. This requires the synchronization of each level that is a certain number of hops away from the source. We call this the *level-synchronized parallel BFS* [6]. Each traversal explores all outgoing edges of the current *frontier*, which is the set of vertices discovered at this level, and finds their *neighbors*, which is the set of unvisited vertices at the next level. We can describe this algorithm using a *frontier* queue Q^F and a *neighbor* queue Q^N, because unvisited vertices w are appended to the *neighbor* queue Q^N for each *frontier* queue vertex $v \in Q^F$ in parallel with the exclusive control at each level (Algorithm 1, lines 7–12), as follows:

$$Q^N \leftarrow \left\{ w \in A(v) \mid w \notin \textit{visited}, v \in Q^F \right\}. \tag{1}$$

B. Hybrid BFS Algorithm of Beamer et al.

The main runtime bottleneck of the level-synchronized parallel BFS (Algorithm 1) is the exploration of all outgoing edges of the current *frontier* (lines 7–12). Beamer et al. [8], [9] proposed a hybrid BFS algorithm (Algorithm 2) that reduced the number of edges explored. This algorithm combines two different traversal kernels: *top-down* and *bottom-up*. Like the level-synchronized parallel BFS, *top-down* kernels traverse *neighbors* of the *frontier*. Conversely, *bottom-up* kernels find the *frontier* from vertices in candidate *neighbors*. In other words, a *top-down* method finds the children from the parent, whereas a *bottom-up* method finds the parent from the children. For a large *frontier*, *bottom-up* approaches reduce the number of edges explored, because this traversal kernel terminates once a single parent is found (Algorithm 2, lines 16–21).

Algorithm 1: Level-synchronized Parallel BFS.

Input : $G = (V, A)$: unweighted directed graph.
s : source vertex.
Variables: Q^F : *frontier* queue.
Q^N : *neighbor* queue.
visited : vertices already visited.
Output : $\pi(v)$: predecessor map of BFS tree.

```
1  π(v) ← −1, ∀v ∈ V
2  π(s) ← s
3  visited ← {s}
4  Q^F ← {s}
5  Q^N ← ∅
6  while Q^F ≠ ∅ do
7      for v ∈ Q^F in parallel do
8          for w ∈ A(v) do
9              if w ∉ visited atomic then
10                 π(w) ← v
11                 visited ← visited ∪ {w}
12                 Q^N ← Q^N ∪ {w}
13     Q^F ← Q^N
14     Q^N ← ∅
```

Table III lists the number of edges explored at each level using a *top-down*, *bottom-up*, and combined *hybrid (oracle)* approach. For the *top-down* kernel, the frontier size $m_{\mathcal{F}}$ at low and high levels is much less than that at mid-levels. In the case of the *bottom-up* method, the frontier size $m_{\mathcal{B}}$ is equal to the number of edges $m = |E|$ in a given graph $G = (V, E)$, and decreases as the level increases. *Bottom-up* kernels estimate all unvisited vertices as candidate *neighbors* Q^N, because it is difficult to determine their exact number prior to traversal, as shown in line 15 of Algorithm 2. This lazy estimation of candidate *neighbors* increases the number of edges traversed for a small *frontier*. Hence, considering the size of the *frontier* and the number of *neighbors*, we combine *top-down* and *bottom-up* approaches in a *hybrid (oracle)*. This loop generally only executes once, so the algorithm is suitable for a small-world graph that has a large *frontier*, such as a Kronecker graph or an R-MAT graph. Table III shows that the total number of edges traversed by the *hybrid (oracle)* is only 3% of that in the case of the *top-down* kernel.

We now explain how to determine a traversal policy (Table II(a)) for the *top-down* and *bottom-up* kernels. The

Algorithm 2: Hybrid BFS algorithm of Beamer et al.

Input : $G = (V, A^{\mathcal{F}},\ A^{\mathcal{B}})$: unweighted directed graph.
s : source vertex.
Variables: Q^F : *frontier* queue.
Q^N : *neighbor* queue.
visited : vertices already visited.
state $\in$ {'*top-down*', '*bottom-up*'} : traversal policy.
Output : $\pi(v)$: predecessor map of BFS tree.

```
1  π(v) ← −1, v ∈ V
2  visited ← {s}
3  Q^F ← {s}
4  Q^N ← ∅
5  state ← 'top-down'
6  while Q^F ≠ ∅ do
7      if state = 'top-down' then
8          for v ∈ Q^F in parallel do
9              for w ∈ A^F(v) do
10                 if w ∉ visited atomic then
11                     π(w) ← v
12                     visited ← visited ∪ {w}
13                     Q^N ← Q^N ∪ {w}
14     else
15         for w ∈ V \ visited in parallel do
16             for v ∈ A^B(w) do
17                 if v ∈ Q^F then
18                     π(w) ← v
19                     visited ← visited ∪ {w}
20                     Q^N ← Q^N ∪ {w}
21                     break
22     state ← traversal_policy(Q^F, Q^N, visited, state)
23     Q^F ← Q^N
24     Q^N ← ∅
```

Table II
SELECTED TRAVERSAL POLICY.

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

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

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

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

evolution of the *frontier* can be divided into a *growing* phase (levels 0–2 in Table III) and a *shrinking* phase (levels 4–6 in Table III). The traversal policy of the hybrid algorithm moves from *top-down* to *bottom-up* in the *growing* phase, and returns from *bottom-up* to *top-down* in the *growing* phase. In this paper, we adopt the same policy for Kronecker graphs and R-MAT graphs. At the intersection between $m_{\mathcal{F}}$ and $m_{\mathcal{B}}$ in the *growing* phase (levels 1 and 2 in Table III), the number of edges traversed by the *top-down* kernel increases greatly. Because of this, the traversal policy of the hybrid algorithm requires careful selection. The algorithm uses the edges traversed by the *top-down* kernel, $m_{\mathcal{F}}$, such that

$$m_{\mathcal{F}} \leftarrow \left|\left\{(v, w) \mid v \in Q^N,\ w \in A^{\mathcal{F}}(v)\right\}\right|, \tag{2}$$

and uses the approximate traversed edges of the *bottom-up* method, $m'_{\mathcal{B}}$, such that

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

where the traversal policy changes if $m_{\mathcal{F}} \cdot \alpha \geq m'_{\mathcal{B}}$. Conversely, edges traversed in the *bottom-up* and *top-down* kernels intersect smoothly in the *shrinking* phase, so this does not require as high a quality of selection as the *growing* phase. Therefore, the algorithm uses the approximate *top-down* traversed edges $m'_{\mathcal{F}}$ such that

$$m'_{\mathcal{F}} \leftarrow \left|Q^N\right| \cdot \text{edgefactor}, \tag{4}$$

and the approximate *bottom-up* traversed edges $m'_{\mathcal{B}}$. Hence, the algorithm uses the exact traversed edges in the *growing* phase and the approximate traversed edges in the *shrinking* phase. In addition, we determine the optimum value of the switching parameters α and β based on the actual execution time (see Section IV-D).

Table III
NUMBER OF EDGES TRAVERSED BY *top-down* ($m_{\mathcal{F}}$), *bottom-up* ($m_{\mathcal{B}}$), AND *hybrid (oracle)* IN A BFS OF A KRONECKER GRAPH (SOURCE VERTEX 19,587,601) WITH SCALE 26 AND EDGEFACTOR 16.

Level	Top-down $m_{\mathcal{F}}$	Bottom-up $m_{\mathcal{B}}$	Hybrid (oracle) $\min(m_{\mathcal{F}}, m_{\mathcal{B}})$
0	2	2,103,840,895	2
1	66,206	1,766,587,029	66,206
2	346,918,235	52,677,691	52,677,691
3	1,727,195,615	12,820,854	12,820,854
4	29,557,400	103,184	103,184
5	82,357	21,467	21,467
6	221	21,240	227
Total	2,103,820,036	3,936,072,360	65,689,631
Ratio	100.00%	187.09%	3.12%

IV. NUMA-OPTIMIZED HYBRID BFS

A. NUMA architecture

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

element in the physical chip. We can obtain useful additional information by checking the processor identifier, package identifier, and core identifier; for example, we can determine which two (or more) threads share the same physical CPU core when using the HT technology.

Table IV
DESCRIPTION OF IDENTIFIERS FOR MAPPING CPU CORES

Identifier	Description
Processor ID	Serial ID (0,1,...) for each CPU core
Package ID	Serial ID (0,1,...) for each CPU socket
Core ID	Serial ID (0,1,...) for physical core in CPU package

In NUMA architecture systems, the memory access time depends on the memory location relative to a processor, and a processor can access its local memory faster than remote (nonlocal) memory (i.e., memory local to another processor or memory shared between processors). In this paper, we propose general techniques of processor affinity and memory binding for NUMA architecture systems. We have already applied similar techniques to graph algorithms for the shortest path problem [2] and mathematical optimization problems [3].

We explain two affinity strategies, *scatter*-type and *compact*-type, for the NUMA architecture. The scatter-type affinity (Fig. 2) distributes OpenMP threads as evenly as possible across the entire system. Conversely, the compact-type affinity binds the (i+1)th OpenMP thread in a free-thread context as closely as possible to the thread context in which the ith OpenMP thread was bound. We select the *scatter*-type affinity because of its localized memory access. Our *scatter*-type affinity binds processors P_k, threads T_k, and local memory RAM_k for each NUMA unit $k \in \{0, 1, ..., \ell - 1\}$ on the ℓ-way NUMA architecture system (see Fig. 2 and Table V). We implement the automatic configuration system for CPU affinity and memory binding on a NUMA architecture and Linux system. This implementation consists of the following three steps:

- Step1: Generate the thread assignment table with scatter-type affinity from the directory structures of Linux device files under the `/sys/devices/system/` (Table V).
- Step2: Set the memory policy to local allocation for packages of threads to be run in Step 3 using `mbind()`.
- Step3: Before executing parallel regions of OpenMP, pin each thread to each processor in the order given in the assignment table using `sched_setaffinity()`.

B. NUMA-optimized Graph Representation

Without loss of generality, we describe the fast BFS algorithm for NUMA architecture systems. It is difficult to achieve high-performance on a NUMA architecture because

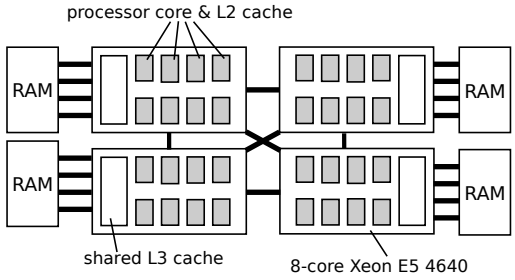


Figure 1. Structure of SandyBridge-EP system.

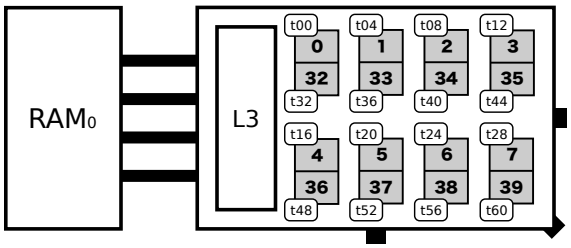


Figure 2. Zeroth NUMA unit (processor P_0, thread T_0, and local memory RAM_0) with *scatter*-type affinity on SandyBridge-EP.

Table V
ASSIGNMENT TABLE OF THREAD, PROCESSOR CORE, AND RAM WITH SCATTER-TYPE AFFINITY FOR EACH NUMA UNIT k ON SANDYBRIDGE-EP.

k	Processors P_k	Threads T_k	RAM
0	00, 01, 02, 03, 04, 05, 06, 07 32, 33, 34, 35, 36, 37, 38, 39	00, 04, 08, 12, 16, 20, 24, 28 32, 36, 40, 44, 48, 52, 56, 60	RAM_0
1	08, 09, 10, 11, 12, 13, 14, 15 40, 41, 42, 43, 44, 45, 46, 47	01, 05, 09, 13, 17, 21, 25, 29 33, 37, 41, 45, 49, 53, 57, 61	RAM_1
2	16, 17, 18, 19, 20, 21, 22, 23 48, 49, 50, 51, 52, 53, 54, 55	02, 06, 10, 14, 18, 22, 26, 30 34, 38, 42, 46, 50, 54, 58, 62	RAM_2
3	24, 25, 26, 27, 28, 29, 30, 31 56, 57, 58, 59, 60, 61, 62, 63	03, 07, 11, 15, 19, 23, 27, 31 35, 39, 43, 47, 51, 55, 59, 63	RAM_3

of the different memory access costs. Agarwal et al. [7] managed the timing of memory access to avoid conflicts between local variables and remote variables, as this can degrade memory access performance. In contrast, we propose a NUMA-optimized algorithm with improved localization of memory access. Our algorithm, which is based on the hybrid algorithm of Beamer et al., 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 remote memory by using the following column-wise partitioning:

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

$$A = \left[A_0 \mid A_1 \mid \cdots \mid A_{\ell-1}\right], \tag{6}$$

and each set of partial vertices V_k on the memory RAM_k is defined by

$$V_k = \left\{ v_j \in V \mid j \in \left[\frac{kn}{\ell}, \frac{k(n+1)}{\ell} \right) \right\}, \tag{7}$$

where the divisor ℓ is set to the number of 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* kernels as follows:

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

$$A_k^{\mathcal{B}}(w) = \{v \in A(w)\}, w \in V_k. \tag{9}$$

Furthermore, the working spaces Q_k^N, visited$_k$, and π_k for partial vertices V_k are allocated to the local memory RAM$_k$ with memory pinned, such that

$$Q_k^N = Q^N \cap V_k, \tag{10}$$

$$\text{visited}_k = \text{visited} \cap V_k, \tag{11}$$

$$\pi_k = \{\pi(v), v \in V_k\}. \tag{12}$$

Algorithms 3 and 4 describe the *top-down* and *bottom-up* methods for the NUMA-optimized BFS algorithm using column-wise partitioning. Both algorithms bind thread group T_k to processor P_k and local memory RAM$_k$ without accessing remote memory, and T_k is processed for local variables in a parallel thread. Our implementation prefers to reference *replicated-csc* code using MPI for a distributed memory environment, but adjacency lists are divided horizontally.

Algorithm 3: NUMA-optimized top-down BFS.

Input : ℓ : number of CPU sockets.
$k = \{0, 1, \cdots, \ell - 1\}$: NUMA unit IDs.
$G = (\{V_k\}, \{A_k^{\mathcal{F}}\})$: unweighted directed graph.
Q^F : *frontier* queue (local copy).
visited $= \{$*visited*$_k\}$: vertices already visited.
$\pi = \{\pi_k\}$: predecessor map of BFS tree.
Output : $Q^N = \{Q_k^N\}$: *neighbor* queue.

```
1  fork()
2  t ← omp_get_thread_num()
3  sched_cpuaffinity(get_processor_id(t), 'scatter')
4  k ← get_package_id(t)
5  T_k ← get_local_threads(k)
6  Q_k^N ← ∅
7  for v ∈ Q^F parallel(T_k) do
8      for w ∈ A_k^F(v) do
9          if w ∉ visited_k atomic then
10             π_k(w) ← v
11             visited_k ← visited_k ∪ {w}
12             Q_k^N ← Q_k^N ∪ {w}
13 join()
```

C. Load Balancing

We show the performance of between-thread load balancing on a NUMA architecture system for our algorithm using column-wise partitioning. For both a Kronecker graph and an R-MAT graph, Table VI shows that the adjacency list A of the input graph $G = (V, E)$ is divided into four approximately equal partial edges $A_k^{\mathcal{F}}$ and $A_k^{\mathcal{B}}$ in each NUMA unit k. This is presumably so that each vertex of the generated edge list was randomized in the generation phase of the Graph500 benchmark. If load balancing is not maintained, we should divide the input graph into subsets with approximately equal numbers of partial edges.

Algorithm 4: NUMA-optimized bottom-up BFS.

Input : ℓ : number of CPU sockets.
$k = \{0, 1, \cdots, \ell - 1\}$: NUMA unit IDs.
$G = (\{V_k\}, \{A_k^{\mathcal{B}}\})$: unweighted directed graph.
Q^F : *frontier* queue (local copy).
visited $= \{$*visited*$_k\}$: vertices already visited.
$\pi = \{\pi_k\}$: predecessor map of BFS tree.
Output : $Q^N = \{Q_k^N\}$: *neighbor* queue.

```
1  fork()
2  t ← omp_get_thread_num()
3  sched_cpuaffinity(get_processor_id(t), 'scatter')
4  k ← get_package_id(t)
5  T_k ← get_local_threads(k)
6  Q_k^N ← ∅
7  for w ∈ V_k \ visited_k parallel(T_k) do
8      for v ∈ A_k^B(w) do
9          if v ∈ Q^F then
10             π_k(w) ← v
11             visited_k ← visited_k ∪ {w}
12             Q_k^N ← Q_k^N ∪ {w}
13             break
14 join()
```

Table VI
LOAD BALANCING (NUMBER OF EDGES) BY COLUMN-WISE PARTITIONING OF ADJACENCY LIST FOR KRONECKER GRAPH AND R-MAT GRAPH WITH SCALE 26 AND EDGEFACTOR 16.

k	Kronecker graph	R-MAT graph
0	$522,450,295$ (24.8%)	$8,322,964$ (24.8%)
1	$525,486,274$ (25.0%)	$8,360,143$ (24.9%)
2	$528,683,697$ (25.1%)	$8,399,616$ (25.1%)
3	$527,221,016$ (25.1%)	$8,443,083$ (25.2%)
Total	$2,103,841,282 \approx 2^{31}$	$33,525,806 \approx 2^{25}$

Fig. 3 shows the efficiency of each level of our hybrid algorithm. This experiment selected the BFS that obtained the median TEPS ratio for a Kronecker graph with scale 26 and edgefactor 16 on a SandyBridge-EP (Table. VII). As shown in Fig. 3, all 64 threads ran at more than 95% efficiency, where 100% represents the fastest thread at each level.

D. Parameter Tuning

In a recent study [9], the switching parameters α and β were determined from numerical results. Similarly, we investigated the variation in BFS performance for different values of α and β for a Kronecker graph and an R-MAT graph on a SandyBridge-EP. We tested 21×21 (α, β) patterns of the form 2^u for some integer $u \in [0, 20]$. The TEPS ratios for a Kronecker graph and R-MAT graph with scale 26 are summarized in Figs. 4(a)–4(c) and Figs. 5(a)–5(c), respectively. From these numerical results, we robustly

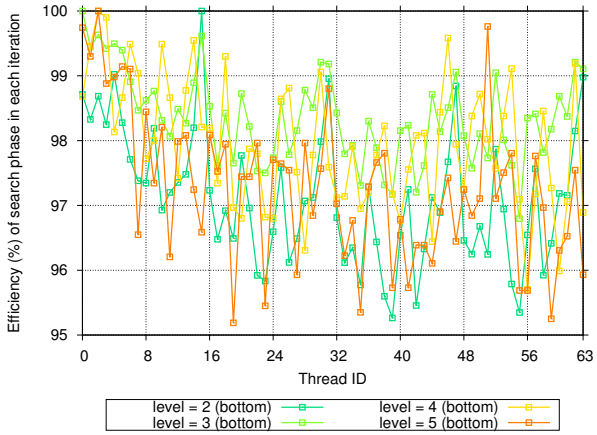


Figure 3. Efficiency of BFS performance (TEPS ratio) for Kronecker graph (scale 26, edgefactor 16) on SandyBridge-EP.

determined the optimum value of (α, β) to be (64,4) for the Kronecker graph and (256,2) for the R-MAT graph.

V. Numerical Results

A. Graph Instances

We measured the performance of our implementation for a Kronecker graph [4] and an R-MAT graph [5] using the reference code of the Graph500 benchmark. Both generators require the scale, edgefactor, and (A, B, C, D) parameters as inputs. They output the 2^{scale} vertices and $2^{\text{scale}} \cdot \text{edgefactor}$ edges that contain some self-loops and some parallel edges. The parameters (A, B, C, D) were set to $(0.57, 0.19, 0.19, 0.05)$ in the manner of the Graph500 benchmark.

B. Machine Environments

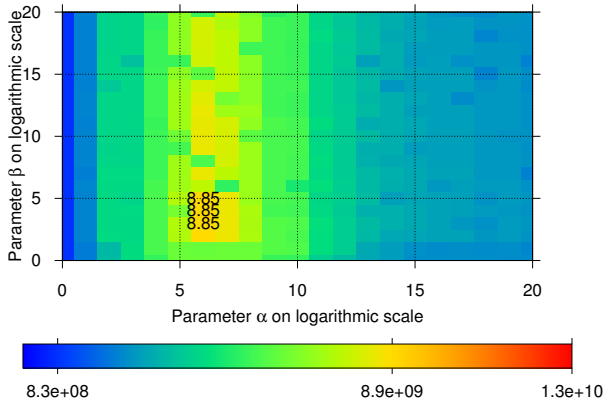
Each NUMA architecture system has ℓ NUMA units, each of which contains t logical cores with multithreading.

Table VII
Machine environments.

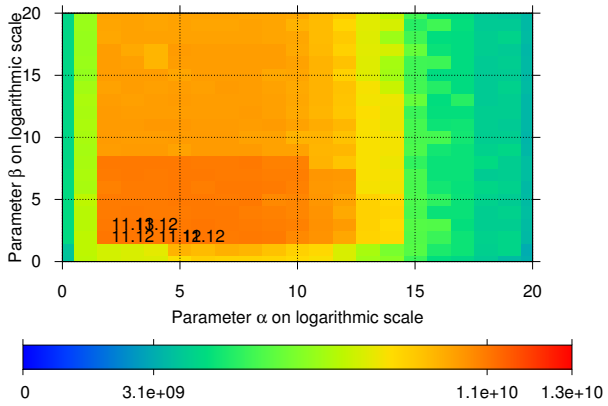
Machine	Processor	$\ell \times t$	OS, compiler
Westmere-EP	Xeon X5670 2.93GHz	2×12	CentOS 5, gcc 4.1.2
Magny-Cours	Opteron 6174 2.20GHz	8×6	Fedora 18, gcc 4.7.2
SandyBridge-EP	Xeon E5-4640 2.40GHz	4×16	CentOS 6, gcc 4.4.7

C. Performance of Top-down, Bottom-up, and Our Hybrid

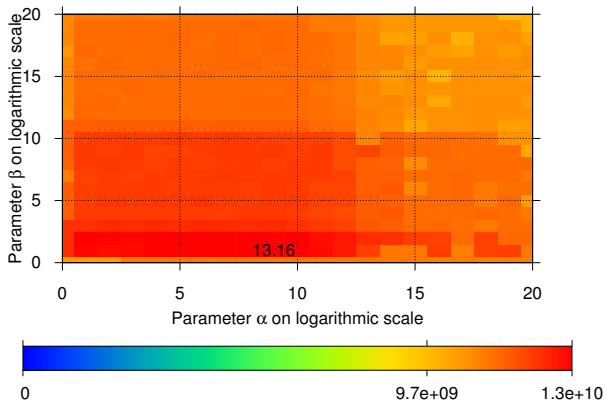
Table VIII shows the CPU time of each level using the *top-down*, *bottom-up*, and hybrid algorithm for the BFS described in Table III. As shown in Table VIII, the performance of the *bottom-up* kernel is considerably faster than that of the *top-down* method for a large *frontier*.



(a) Minimum TEPS $[8.3 \times 10^8, 8.9 \times 10^9]$

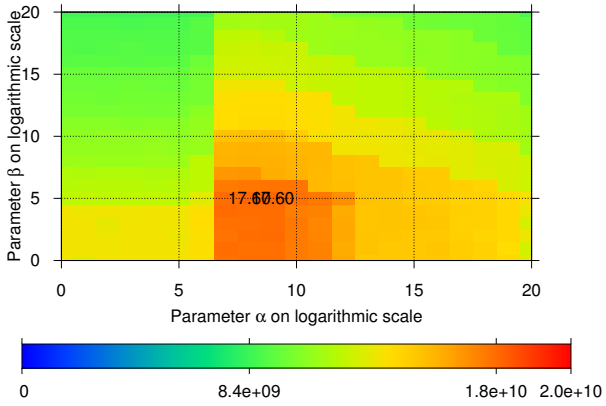


(b) Median TEPS $[3.1 \times 10^9, 1.1 \times 10^{10}]$

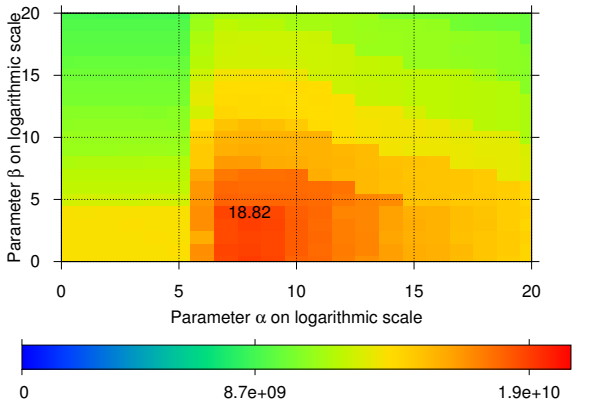


(c) Maximum TEPS $[9.7 \times 10^9, 1.3 \times 10^{10}]$

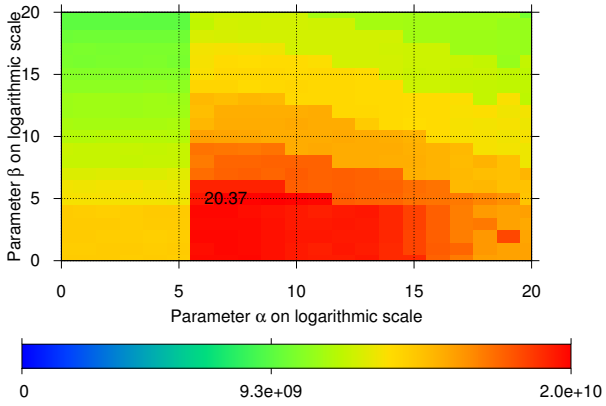
Figure 4. Optimization of (α, β) using TEPS ratio for a Kronecker graph (scale 26, edgefactor 16) on SandyBridge-EP.



(a) First quartile TEPS $[8.4 \times 10^9, 1.8 \times 10^{10}]$



(b) Median TEPS $[8.7 \times 10^9, 1.9 \times 10^{10}]$



(c) Maximum TEPS $[9.3 \times 10^9, 2.0 \times 10^{10}]$

Figure 5. Optimization of (α, β) using TEPS ratio for an R-MAT graph (scale 26, edgefactor 16) on SandyBridge-EP.

Table VIII
CPU TIME OF EACH LEVEL OF THE TOP-DOWN, BOTTOM-UP, AND HYBRID METHODS ($\alpha = 64$, $\beta = 2$) FOR A KRONECKER GRAPH WITH SCALE 26 (SOURCE VERTEX 19,587,601) ON SANDYBRIDGE-EP.

Level	Top-down	Bottom-up	Our hybrid
init.	9.3 ms	9.9 ms	9.3 ms
0	0.1 ms	286.9 ms	0.1 ms (T)
1	1.4 ms	237.0 ms	2.2 ms (T)
2	200.7 ms	46.6 ms	46.0 ms (B)
3	921.9 ms	27.1 ms	26.9 ms (B)
4	76.0 ms	7.1 ms	6.8 ms (B)
5	0.5 ms	7.0 ms	5.1 ms (B)
6	0.1 ms	6.9 ms	0.1 ms (T)
Total	1210.0 ms	628.5 ms	96.5 ms
Ratio	0.90 GTEPS	1.71 GTEPS	11.12 GTEPS

D. Strong Scaling

We now demonstrate the scalability of our hybrid algorithm on NUMA architecture systems. As shown in Table IX(a) and Fig. 6(a), the strong scaling of our hybrid with 64 threads is approximately 28 times faster than sequential computation for a Kronecker graph with scale 26 and edgefactor 16. When the edgefactor was set to 64, the scalability of our hybrid algorithm was higher than for an edgefactor of 16. In contrast, Table IX(b) and Fig. 6(b) show superlinear scaling for an R-MAT graph. This means that the number of repeated parallel edges increased with the edgefactor of the R-MAT graph. However, the number of edges actually traversed is not as large as the increment of edges in some BFSs. Our implementation runs faster with 64 threads than for any other number (i.e., 1, 2, 4, 8, 16, or 32 threads), and is thus able to handle multithreading. Fig. 8(a) shows the scalability of our hybrid in terms of its strong scaling for each NUMA system (i.e., Westmere-EP, Magny-Cours, and SandyBridge-EP).

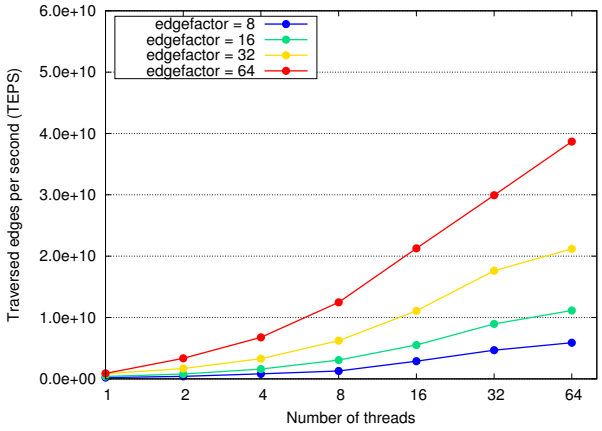
Table IX
STRONG SCALING OF OUR HYBRID ON SANDYBRIDGE-EP

(a) Kronecker graph (scale 26, edgefactor 16 and 64)

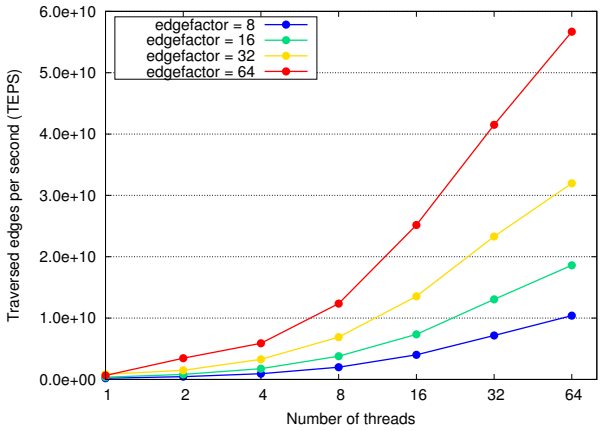
Threads	GTEPS	Speedup	GTEPS	Speedup
	edgefactor = 16		*edgefactor = 64*	
1	0.399	1.0	0.886	1.0
2	0.798	2.0	3.346	3.8
4	1.596	4.0	6.764	7.6
8	3.061	7.7	12.479	14.1
16	5.531	13.9	21.282	24.0
32	8.959	22.5	29.939	33.8
64	11.149	27.9	38.684	43.7

(b) R-MAT graph (scale 26, edgefactor 16 and 64)

Threads	GTEPS	Speedup	GTEPS	Speedup
	edgefactor = 16		*edgefactor = 64*	
1	0.364	1.0	0.647	1.0
2	0.814	2.2	3.465	5.4
4	1.750	4.8	5.898	9.1
8	3.790	10.4	12.363	19.1
16	7.341	20.1	25.179	38.9
32	13.057	35.8	41.512	64.2
64	18.604	51.1	56.686	87.6

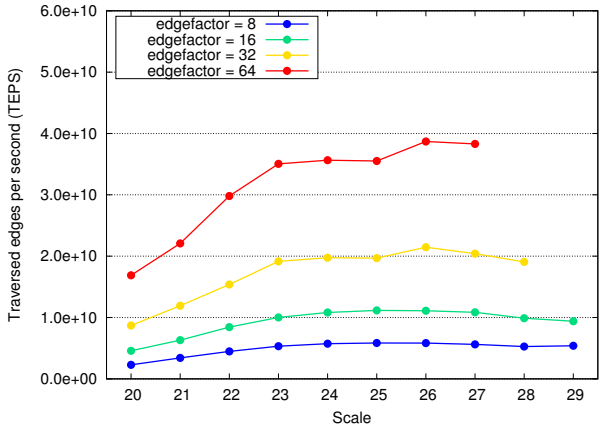


(a) Kronecker graph (scale 26)

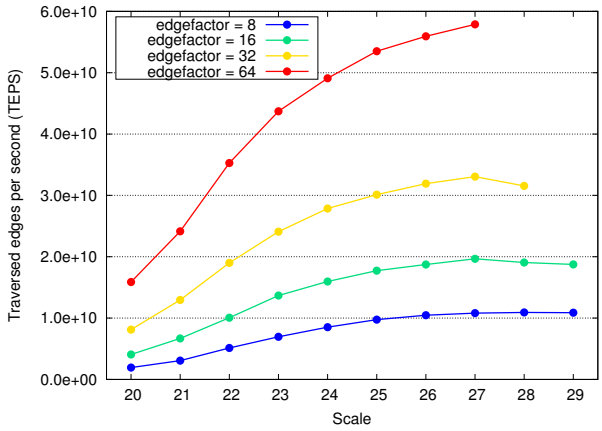


(b) R-MAT graph (scale 26)

Figure 6. Scalability (strong scaling) of our hybrid on SandyBridge-EP.



(a) Kronecker graph



(b) R-MAT

Figure 7. Performance (TEPS ratio) of our hybrid on SandyBridge-EP

E. Performance Variation with Problem Size

Section V-D presented numerical results for a Kronecker graph and R-MAT graph with scale 26 and edgefactor 16. Figs. 7(a) and 7(b) illustrate how the performance of our algorithm varied with the scale and edgefactor of the Kronecker graph and R-MAT graph, respectively. The blanks in the figures indicate points at which the system had insufficient memory space to execute the algorithm. For both the Kronecker graph and the R-MAT graph, the peak TEPS of our hybrid algorithm was attained with a scale of 25, 26, or 27 for edgefactors of 8, 16, 32, and 64. Fig. 8(b) compares the performance of our hybrid on different NUMA systems with respect to scale.

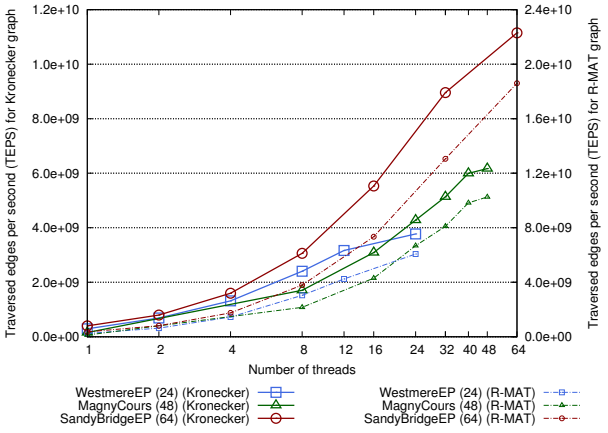
Table X compares the reference code (Graph500 version 2.1.4) and our algorithm for a Kronecker graph and R-MAT graph with edgefactor 16 on the SandyBridge-EP. The speedup ratio of our algorithm with respect to the reference code generally increases with scale for the Kronecker graph. Each blank in the table indicates that the reference code generated a serious error and was aborted.

Table X
PERFORMANCE (GTEPS AND SPEEDUP RATIO) OF REFERENCE CODE (REF.) AND OUR HYBRID ALGORITHM (OURS) ON SANDYBRIDGE-EP

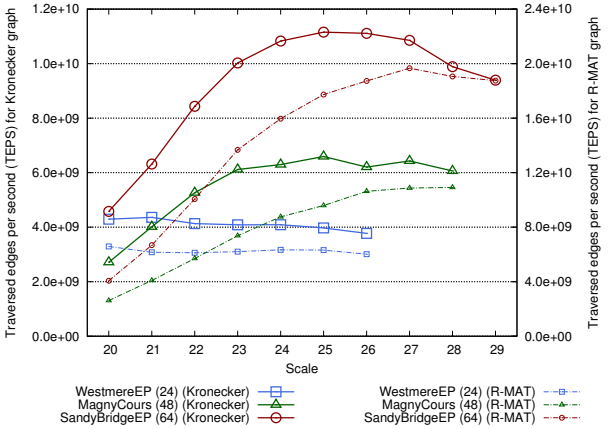
	Kronecker			**R-MAT**		
Scale	*Ref.*	*Ours*	*Speedup*	*Ref.*	*Ours*	*Speedup*
20	0.358	4.578	12.8	0.034	4.073	118.2
21	0.233	6.317	27.1	0.070	6.684	96.0
22	0.101	8.433	83.7	0.127	10.063	79.1
23	0.129	10.023	77.6	0.248	13.673	55.2
24	0.123	10.829	88.3	0.466	15.960	34.2
25	0.107	11.155	104.4	0.737	17.733	24.1
26	0.097	11.149	114.9	1.087	18.727	17.2
27	0.087	10.854	124.5	–	19.659	–
28	–	9.887	–	–	19.060	–
29	–	9.393	–	–	18.754	–

VI. CONCLUSION

We adapted the algorithm of Beamer et al. to NUMA systems, such as the SandyBridge-EP, by applying a column-wise partitioning of the adjacency list while considering the NUMA architecture. In our algorithm, local threads traverse each adjacency list allocated on local RAM. This algorithm, which avoids the overhead of remote RAM access, achieved



(a) Scalability for Kronecker graph and R-MAT graph (scale 26, edgefactor 16)



(b) Performance for Kronecker graph and R-MAT graph (edgefactor 16)

Figure 8. Scalability (strong scaling) and performance (TEPS ratio) of our hybrid on Westmere-EP, Magny-Cours, and SandyBridge-EP.

a high level of efficiency in several numerical simulations. The implementation achieved 8.15 GTEPS and ranked 25th (the fastest of the single-node algorithms) on the Graph500 list in June 2012. An updated implementation achieved 10.50 GTEPS and 11.15 GTEPS, and ranked 52nd and 57th on the Graph500 list in November 2012 and June 2013, respectively (the fastest of the CPU-based single-node algorithms). In addition, our implementation dominated the rankings (first through fifth) in the first Green Graph500 list in June 2013.

ACKNOWLEDGMENT

This research was supported by the Core Research for Evolutional Science and Technology (CREST) program of the Japan Science and Technology Agency (JST).

REFERENCES

[1] U. Brandes, "A faster algorithm for betweenness centrality," J. Math. Sociol., vol. 25, no. 2, pp. 163-177, 2001.

[2] 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, vol. 54, no. 4, pp. 259–280, 2011.

[3] K. Fujisawa, T. Endo, H. Sato, M. Yamashita, S. Matsuoka and M. Nakata, "High-Performance General Solver for Extremely Large-scale Semidefinite Programming Problems," Proceedings of the 2012 ACM/IEEE Conference on Supercomputing, SC'12, 2012.

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

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

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

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

[8] 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, 2011, UCB/EECS-2011-117.

[9] S. Beamer, K. Asanović, and D. A. Patterson, "Direction-optimizing breadth-first search," Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC12), IEEE Computer Society, 2012, no. 12.

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

[11] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC11), ACM, 2011, no. 65.

[12] 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," Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC12), IEEE Computer Society, 2012, no. 13.

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