NUMA-aware Scalable Graph Traversal on SGI UV Systems

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

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

Eng Lim Goh, John Baron, Atsushi Sugiura
Silicon Graphics International Corp.
900 North McCarthy Blvd., Milpitas, CA, USA
{englim,jbaron,asugiura}@sgi.com

Takashi Uchiyama
SGI Japan, Ltd.
Yebisu Garden Place Tower 31F, 4-20-3 Ebisu, Shibuya-ku, Tokyo
uchiyama@sgi.com

ABSTRACT

Breadth-first search (BFS) is one of the most fundamental processing algorithms in graph theory. We previously presented a scalable BFS algorithm based on Beamer's direction-optimizing algorithm for non-uniform memory access (NUMA)-based systems, in which the NUMA architecture was carefully considered. This paper presents our new implementation that reduces remote memory access in a top-down direction of direction-optimizing algorithm. We also discuss numerical results obtained on the SGI UV 2000 and UV 300 systems, which are shared-memory supercomputers based on a cache coherent (cc)-NUMA architecture that can handle thousands of threads on a single operating system. Our implementation has achieved performance rates of 219 billion edges per second on a Kronecker graph with 2^{34} vertices and 2^{38} edges on a rack of an SGI UV 300 system with 1,152 threads. This result exceeds the fastest entry for a shared-memory system on the current Graph500 list presented in November 2015, which includes our previous implementation.

CCS Concepts

•**Mathematics of computing** → **Graph algorithms;**
•**Theory of computation** → **Shared memory algorithms;**

Keywords

Graph algorithm, NUMA-aware, Graph500 benchmark

1. INTRODUCTION

Breadth-first search (BFS) is one of the most important and fundamental graph algorithms. 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 algorithm, but also works as a subroutine in applications that determine the maximum flow [6, 7], connected components [5], graph centrality [3, 8], and clustering [10]. The well-known BFS algorithm [5] that uses the first-in first-out (FIFO) queue, theoretically 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 is optimal for theoretical purposes, but there is an actual need for efficient graph processing for large-scale real-world networks. Theoretical complexity analysis alone is insufficient because large-scale BFS computations require a significant amount of memory to enable multiple memory accesses over a wide memory space.

In this paper, we present an efficient graph traversal algorithms in which the non-uniform memory access (NUMA) architecture was carefully considered. Table 1 shows related work and our algorithms and implementations on a four-socket Xeon server. Our implementations–namely, BD13 [20], ISC14 [21], HPCS15-DG [22]– used different graph structures in the top-down and bottom-up directions to cut off remote memory access in each edge traversal direction. These require an all-gather operation of vertices traversed at the current level for the next level, which includes many remote memory accesses. Although this operation consumes scant CPU time on a NUMA system with a few CPU sockets, it can result in bottleneck points on large-scale shared-memory supercomputers with many CPU sockets. We described the HPCS15-SG [22] implementation for such a system, which used the same graph structure in both directions. The top-down direction search of this implementation is the same as that of Agarwal's algorithm [1]. Table 2 lists the typical numerical results we obtained in terms of traversed edges per second (TEPS) on shared-memory supercomputers and on a large memory server–specifically, SGI UV 300, SGI UV 2000, and HP SuperdomeX. Our previous implementation, HPCS15-SG, achieved 174 GTEPS on a Kronecker graph with SCALE 34 on a shared-memory SGI UV 2000 supercomputer with 1,280 threads, which is the fastest entry on a shared-memory single-node system in the November 2014, July 2015, and November 2015 Graph500 lists. Our present implementation, in which remote edge traversal pruning (shown in Table 1 as TD-Prun.) is applied to Agarwal's top-down algorithm to reduce remote memory access, achieves 152 GTEPS for SCALE 34 on SGI UV 2000 with 1280 threads and 219 GTEPS for SCALE 34 on UV 300. The new results presented in this paper exceed the

HPGP'16, May 31-30, 2016, Kyoto, Japan

DOI: http://dx.doi.org/10.1145/2915516.2915522

Table 1: TEPS scores of Related work on 4-socket Intel Xeon servers.

Year	Implementation	SCALE	TEPS	Top-down	Dir. opt.	NUMA-graph	Adj.Sort.	Vtx.Sort.	TD-Prun.
2010	Reference [16]	27	0.1 G	✓					
2010	Agarwal [1]	20	1.3 G	✓		A^B			
2012	Beamer [2]	28	5.1 G		✓				
2013	BD13 [20]	26	11.2 G		✓	(A^F, A^B)			
2014	ISC14 [21]	27	29.0 G		✓	(A^F, A^B)	✓		
2015	HPCS15-DG [22]	27	41.8 G		✓	(A^F, A^B)	✓	✓	
2015	HPCS15-SG [22]	30	28.6 G		✓	(A^B, A^B)	✓	✓	
2016	Our present implementation	30	31.3 G		✓	(A^B, A^B)	✓	✓	✓

fastest entry for a shared-memory system in the current, November 2015, Graph500 list. In addition, we investigated the network topology details of SGI UV systems in terms of memory bandwidth achieved using the STREAM benchmark. We also compared memory bandwidth using the STREAM benchmark and graph traversal performance (TEPS score) using the Graph500 benchmark.

Table 2: TEPS scores on SGI UV systems.

System	#Threads	#Sockets	SCALE	TEPS
Our previous implementation HPCS15-SG [22]				
SGI UV 2000	640	64	32	131 G
SGI UV 2000	1,280	128	33	174 G
HP SuperdomeX	480	16	33	128 G
This paper				
SGI UV 2000	640	64	33	152 G
SGI UV 300	1,152	32	34	219 G

2. PRELIMINARIES

2.1 Graph500 Benchmark

The Graph500 benchmark[1] is designed to measure computational performance for applications that require an irregular memory access pattern. It is based on a score of TEPS, which is computed using the generated edge list and the output of the BFS [16]. The Green Graph500 benchmark[2] is designed to measure the energy efficiency of a computer in terms of TEPS per Watt [11]. These lists are updated biannually since their introduction in 2010. Both benchmarks must perform the following steps: (1) **Generation:** This step generates the edge list of a Kronecker graph [14] with 2^{scale} vertices and $2^{scale} \cdot edgefactor$ edges by *scale* times the Kronecker products of an initiator matrix $\begin{pmatrix} 0.57 & 0.19 \\ 0.19 & 0.10 \end{pmatrix}$, where the *scale* and the *edgefactor* are input parameters. (2) **Construction (timed):** This step constructs the graph representation from the edge list obtained in Step 1. (3) **BFS iterations (timed):** This step executes 64 BFSs from different source vertices and computes the median TEPS score of 64 TEPS scores.

2.2 Parallel BFS

First, 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. 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 the source vertex $\pi(s)$ points itself to s.

The well-known textbook algorithm for BFS is not suitable for parallelism, which uses the FIFO queue. Therefore, we use Algorithm 1 (called Level-synchronized Breadth-first search), which utilizes two queues: *current queue* CQ and *next queue* NQ. In this algorithm, we assume that an input graph $G = (V, A^F)$, based on an adjacency vertex list A^F represents a directed graph, where an adjacency list $A^F(v)$ contains the adjacency vertices w of outgoing edges $(v, w) \in E$ for each vertex $v \in V$. If an input graph is undirected, it uses (v, w) and (w, v) edges instead of $(v, w) \in E$ edges. This algorithm starts with the current queue CQ as the source s. At each level k, this algorithm finds unvisited adjacency vertices $A^F(v), v \in$ CQ that are connected to the current queue CQ, and appends them to the next frontier NQ for level k+1. After the edge traversal, NQ becomes the current queue CQ for the next level. The algorithm terminates when the frontier is empty. For consistency in the deepest loop, this algorithm requires atomic operations, which call the same number of edges. In general, this is guaranteed to have a high cost and will therefore be a performance bottleneck.

Algorithm 1: Level-synchronized Breadth-first search

```
   Input  : Digraph G = (V, A^F), vertex s.
   Data   : current queue CQ, next queue NQ, and visited
            vertices VS.
   Output: predecessor π(v), ∀v ∈ V.
 1 π(v) ←⊥,  ∀v ∈ V\{s}
 2 π(s) ← s
 3 VS ← {s}
 4 CQ ← {s}
 5 NQ ← ∅;
 6 while CQ ≠ ∅ do
 7 |   NQ ← Top-down(G, CQ, VS, π)
 8 |   swap(CQ, NQ)
 9 Procedure Top-down(G, CQ, VS, π)
10 |   NQ ← ∅
11 |   for v ∈ CQ in parallel do
12 |   |   for w ∈ A^F(v) do
13 |   |   |   if w ∉ VS atomic then
14 |   |   |   |   π(w) ← v
15 |   |   |   |   VS ← VS ∪ {w}
16 |   |   |   |   NQ ← NQ ∪ {w}
17 |   return NQ
```

Beamer et al. [2] proposed a direction-optimizing algo-

[1]Graph500 benchmark: http://www.graph500.org.
[2]Green Graph500 benchmark: http://green.graph500.org.

rithm for BFS (Algorithm 2) that reduces the number of edges explored. Like Algorithm 1, this algorithm performs a traversal procedure (lines 7–10) and swaps NQ and CQ (line 11) at each level. This algorithm has two different traversal directions, *top-down* and *bottom-up*, from which it chooses one according to the size of the current queue $|\deg_G v, v \in \mathrm{CQ}|$. The former traverses the next queue NQ from the current queue CQ, whereas the latter finds the *frontier* CQ from all unvisited vertices $V \backslash \mathrm{VS}$ as candidate neighbors. Table 3 shows how the traversal direction is determined for the top-down and bottom-up approaches (line 7). The traversal direction moves from top-down to bottom-up in the *growing* phase $|\mathrm{CQ}| < |\mathrm{NQ}|$, and returns from bottom-up to top-down in the *shrinking* phase $|\mathrm{CQ}| \geq |\mathrm{NQ}|$. The computational complexities are $O(m)$ for the top-down direction and $O(m \cdot \mathrm{diam}_G)$ for the bottom-up direction, where m is the number of edges and diam_G is the diameter of the given graph. The direction-optimizing algorithm that combines these algorithms has $O(m \cdot \mathrm{diam}_G)$ complexity; however, it works well experimentally.

Algorithm 2: Direction-optimizing Breadth-first search

Input : Digraph $G = (V, A^F,\ A^B)$, vertex s.
Data : frontier queue CQ, next queue NQ, and visited vertices VS.
Output: Predecessor map $\pi(v)$, $\forall v \in V$.

```
1  π(v) ←⊥, ∀v ∈ V\{s}
2  π(s) ← s
3  VS ← {s}
4  CQ ← {s}
5  NQ ← ∅
6  while CQ ≠ ∅ do
7      if use_TopDown(G, CQ, NQ, VS) then
8          NQ ← Top-down(G, CQ, VS, π)
9      else
10         NQ ← Bottom-up(G, CQ, VS, π)
11     swap(CQ, NQ)
12 Procedure Bottom-up(G, CQ, VS, π)
13     NQ ← ∅
14     for w ∈ V \ VS in parallel do
15         for v ∈ A^B(w) do
16             if v ∈ CQ then
17                 π(w) ← v
18                 VS ← VS ∪ {w}
19                 NQ ← NQ ∪ {w}
20                 break
21     return NQ
```

2.3 NUMA-aware Parallel BFS

2.3.1 NUMA-aware partitioned graph representation

Our NUMA-optimized algorithms, which are based on Beamer et al.'s direction-optimizing algorithm [2], use the 1D partitioning for sets of vertices and edges to improve access to local memory [20]. Polymer, another NUMA-aware implementation for graph analytics, uses a similar graph representation to reduce excessive random remote accesses [23].

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

$$\begin{aligned} V_k &= \left\{ v_j \mid j \in \left[\frac{n}{\ell} \cdot k, \frac{n}{\ell} \cdot (k+1) \right) \right\}, \\ E_k &= \{(v,w) \mid ((v,w) \in E) \wedge (v \in V) \wedge (w \in V_k)\} \end{aligned} \tag{1}$$

Table 3: Number of edges traversed for each traversal direction in the BFS of a Kronecker graph with SCALE 26 and edgefactor 16.

Level	Top-down m_F	Bottom-up m_B	Dir. Opt. (best) $\min(m_F, m_B)$
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	221
Total	2,103,820,036	3,936,072,360	65,689,631
Ratio	100.00%	187.09%	3.12%

where n is the number of vertices and the divisor ℓ is set to the number of NUMA nodes (CPU sockets). Our implementations use partial adjacency lists $A_k^F(v), v \in V$ for the top-down direction and $A_k^B(w), w \in V_k$ for the bottom-up direction on the k-th NUMA node as follows:

$$\begin{aligned} A_k^F(v) &= \{w \mid (v,w) \in E_k\}, v \in V, \\ A_k^B(w) &= \{v \mid (v,w) \in E_k\}, w \in V_k. \end{aligned} \tag{2}$$

As outlined above, the working spaces NQ, VS, and π in Algorithms 1 and 2 are partitioned into NQ_k, VS_k, and π_k using corresponding partial vertices V_k and allocated to the local memory on the k-th NUMA node with the memory pinned. In contrast to the NQ, VS, and π, the current queue CQ is duplicated into CQ_k, which is allocated to local memory on the k-th NUMA node.

2.3.2 Adjacency list sorting

The bottom-up procedure checks that each unvisited vertex connects to the frontier vertices that are included in the current queue. Consequently, the number of loops in the bottom-up procedure depends on the order of each adjacency list for each unvisited vertex. It is difficult to obtain the optimal ordering for the adjacency vertex list, therefore we use a heuristic that constructs an adjacency vertex list $A(v)$ for each vertex $v \in V$, which is then sorted by the out-degree [21]. Table 4 compares the number of traversed edges for each level in the top-down and the bottom-up directions for each order; *Descending order* and *Ascending order*. The table shows that most of the traversed edges are concentrated in Level-2 and the number of traversed edges is affected by the order.

2.3.3 Vertex index sorting

We use a vertex index sorting technique [22] to improve the locality of access to working space VS in the top-down and CQ in the bottom-up (mainly bottleneck component) directions, which is similar to that in [18]. The current queue CQ in the bottom-up direction, is implemented using a bitmap structure, which represents the set for each vertex as one bit. If the corresponding vertices are in a set, then the bit is 1; otherwise, the bit is 0. Because the number of accesses by each element is equal to the in-degree $\deg_G^{\mathrm{in}}$ of corresponding vertex $v \in V$, if the elements frequently used are located close together in memory, a cache memory works well. To construct the first graph, our technique constructs new vertex indices $\{0, 1, ..., n-1\}$ as follows:

$$\deg_G^{\mathrm{in}}(v_0) \geq \deg_G^{\mathrm{in}}(v_1) \geq ... \geq \deg_G^{\mathrm{in}}(v_{n-1}). \tag{3}$$

Table 4: Number of traversed edges in a BFS for Kronecker Graph with SCALE 27.

		Descending order		Ascending order	
Level	**Top-down**	**Bottom-up**	**Dir.Opt.** min(Td, Bu)	**Bottom-up**	**Dir.Opt.** min(Td, Bu)
0	22	4,223,250,243	22	4,223,039,317	22
1	239,930	3,258,645,723	239,930	4,063,345,725	239,930
2	1,040,268,126	**83,878,899**	**83,878,899**	**848,743,124**	**848,743,124**
3	3,145,608,885	19,616,130	19,616,130	19,935,737	19,935,737
4	37,007,608	139,606	139,606	139,868	139,868
5	98,339	41,846	41,846	41,846	41,846
6	260	41,586	260	41,586	260
Total	4,223,223,170	7,585,614,033	103,916,693	9,155,287,203	869,100,787
%	100%	179.6%	2.5%	216.8%	20.6%

3. NUMA-AWARE COMPUTATION

At present, major systems are designed based on the NUMA and cc-NUMA architectures. On such NUMA systems, each processor has a local memory, and are connected via an interconnect such as the Intel QPI, AMD HyperTransport, or SGI NUMAlink. Each thread running on a processor core can access local memory faster than remote (non-local) memory on a NUMA system. Table 5 lists the NUMA systems used in this paper.

The performance of BFS depends on the speed of memory access, because the complexity of memory accesses is greater than that of computation. From these results, we considered the placements of running threads and referenced data to improve the performance on NUMA systems. NUMA-aware algorithms and data structures are also efficient for other graph algorithms on multi-socket multi-core NUMA systems. Galois [15] achieved high-scalability by applying NUMA-aware memory allocation, scheduling, and barrier routines. Polymer [23] used efficient graph representation to reduce random remote memory access, which is similar to our graph representation [20].

3.1 Memory bandwidth between NUMA nodes

In this section, we first investigate the characteristics of NUMA system in terms of memory bandwidth using the TRIAD operation of the STREAM benchmark[3] and the `numactl` command. The TRIAD operation computes $\mathbf{a} \leftarrow \mathbf{b} + r \cdot \mathbf{c}$ using three vectors $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbf{R}^n$ with n elements (defined as `STREAM_ARRAY_SIZE` in `stream.c`) and a scalar $r \in \mathbf{R}$, whose element holds a double precision floating point number (8 bytes per element). Fig. 2a shows the bandwidths between two NUMA nodes on SB4 using the `numactl` command. We specified the placement of running threads and referenced data using the `--cpunodebind` option for CPU binding and the `--membind` option for memory binding of `numactl` command, shown in Fig. 1. From this result, the local memory access (24.3 – 24.6 GB/s) is approximately eight times faster than the remote memory access (2.9 – 3.4 GB/s). In addition, we describe the obtained bandwidths in a punch figure of SB4, shown in Fig. 2b.

Similarly, Figs. 3a and 3b show the bandwidths between two NUMA nodes on one rack of SGI UV 2000 and SGI UV 300 systems, respectively. Tables 6a and 6b summarize the average memory bandwidth for each topology. The network topology of the UV 2000 is based on the hypercube of eight compute nodes that connect via the NUMAlink 6 interconnect. Each point of the hypercube that corresponds a compute node that has 2 CPU sockets (2 NUMA nodes),

[3]STREAM benchmark: https://www.cs.virginia.edu/stream/

```
wget \
 https://www.cs.virginia.edu/stream/FTP/Code/stream.c
icc -O2 -fopenmp -DSTREAM_ARRAY_SIZE=100000000 \
                                    -o stream stream.c
SOCKETS=$(seq 0 31)
THREADS=36
for i in $SOCKETS; do
  for j in $SOCKETS; do
    OMP_NUM_THREADS=$THREADS \
      numactl --cpunodebind=$i --membind=$j ./stream
done; done
```

Figure 1: shell script for memory bandwidth test on UV 300.

connects to a point of another hypercube via the NUMAlink 6 router. As shown in Table 6a, the UV 2000 has the complex topology.

On the other hand, the compute chassis of UV 300 that have four sockets, and connect directly to each other via the NUMAlink 7 interconnect. Therefore, the network topology of UV 300 is simpler than that of UV 2000. Although many characteristics exist on the network topology, we surmise that it is safe to assume that both systems have fast local memory and slow remote memory simply due to the performance gaps between them.

3.2 ULIBC

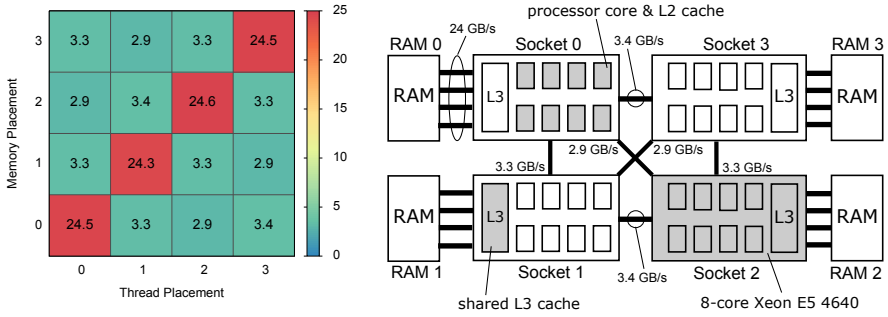
In this section, we propose a general management approach for processor and memory affinities on NUMA systems. Previous proposals include the Portable Hardware Locality (hwloc) [4], the Likwid [17], Thread Affinity Interface of Intel compiler [13], and OpenUH compiler [12]. However, to the best of our knowledge there is no library for obtaining the position of each running thread, such as the CPU socket index, physical-core index in each CPU socket, or thread index in each physical-core. Consequently, we developed a management library for processor and memory affinities, called ULIBC. ULIBC supports many operating systems (although we have only confirmed Linux, Solaris, and AIX) and is available at

https://bitbucket.org/yuichiro_yasui/ulibc.

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 [20]. We have already applied ULIBC to graph algorithms for shortest paths and centrality [19], BFS [20, 21, 22], and mathematical optimization problems [9]. Fig. 4 is a memory bandwidth plot obtained using the TRIAD operation of the STREAM benchmark, to which was applied pinning for the running threads and the vector data using ULIBC.

Table 5: NUMA systems

System	CPU name (LLC size)	Sockets × Cores × SMT	RAM	Compiler
SB4	Xeon E5-4640 (20 MB)	4 × 8 × 2	512.0 GB	ICC-15.0.1
HP Superdome X	Xeon E7-2890 v2 (37.5 MB)	16 × 15 × 2	12.0 TB	GCC-4.3.4
SGI UV 2000	Xeon E5-4650 v2 (25 MB)	256 × 10 × 1	64.0 TB	ICC-14.0.0
SGI UV 300	Xeon E7-8890 v3 (45 MB)	32 × 18 × 2	16.0 TB	ICC-16.0.0



(a) Bandwidths (b) SB4 (4-socket Intel Xeon E5-4640) system

Figure 2: Bandwidth score GB/s between arbitrary NUMA nodes.

Table 6: Breakdown of memory bandwidth GB/s (average) between arbitrary two NUMA nodes.

(a) UV 2000 with 64 CPU sockets (One rack)

No.	Location	GB/s	#pairs
1	Node: NUMA local	32.47	64
2	Node: NUMA remote	6.85	64
3	Cube: NUMAlink6 1 hop	5.55	640
4	Cube: NUMAlink6 2 hops	4.78	256
5	Block: NUMAlink6 2 hops	4.45	384
6	Block: NUMAlink6 3 hops	3.84	1920
7	Block: NUMAlink6 4 hops	3.43	768
8	Inter-Block: NUMAlink6 3 hops	–	0
9	Inter-Block: NUMAlink6 4 hops	–	0
10	Inter-Block: NUMAlink6 5 hops	–	0
total		–	4096

(b) UV 300 with 32 CPU sockets (One rack)

No.	Location	GB/s	#pairs
1	Node: NUMA local	56.34	32
2	Node: NUMA remote (QPI) 1 hop	14.16	64
3	Node: NUMA remote (QPI) 2 hops	12.39	32
4	Inter-Node: NUMAlink7 1 hop	5.90	896
total		–	1024

This bandwidth improves linearly with increasing number of sockets. This result corresponds to the sum of the memory access performance for local memory.

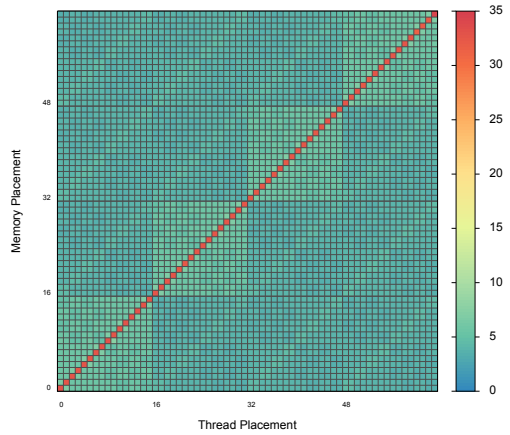
4. REMOTE EDGE TRAVERSAL PRUNING

In this subsection, we explain our new implementation based on HPCS15-SG [22]. Algorithm 3 describes the top-down direction applied for the pruning of remote edge traversal in Agarwal's algorithm [1] to reduce remote memory access. Agarwal's algorithm appends all adjacency vertices owned by another NUMA node to the corresponding socket queue (SQ), when it finds that other owner. In contrast, Algorithm 3 appends each vertex only once using bitmap F with n bits, where n is the number of vertices of the given graph, shown in lines 11–13. Fig. 5 and Table 7 show the number of traversed edges of each access pattern at each level for a Kronecker graph with SCALE29 on the four-socket Xeon server (SB4). This result was obtained using Algorithm 3 at each level (with no bottom-up). This figure shows that this pruning technique significantly reduces the number of remote memory accesses.

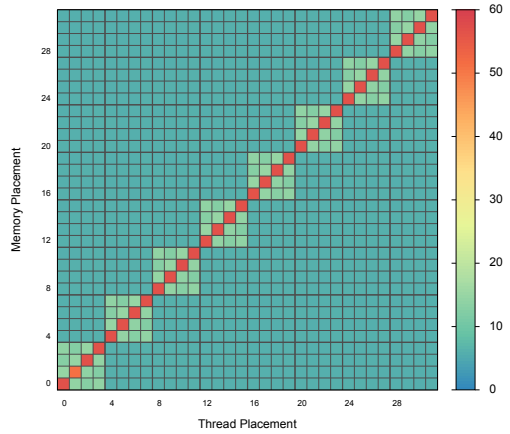
The respective total for local access case, pruned remote access case, and remote access case are 4,246,735,792 edges, 12,285,669,601 edges, and 454,592,619 edges. Compared with previous implementations that access $\frac{1}{\ell}$ local memory and $\frac{\ell-1}{\ell}$ remote memory on the ℓ-socket server, our new algorithm improves this ratio to 97.3% for local edges and 2.7% for remote edges. In addition, this algorithm can apply a direction-optimizing algorithm easily, using current queue CQ of the bottom-up direction as F. Unfortunately, this improvement might not be as effective as in this example, owing to the fact that our implementation switches direction from the top-down to the bottom-up for large frontiers (Levels 2, 3, 4, and 5 in this example) of a Kronecker graph.

Table 7: Classification of number of traversed edges obtained for the top-down algorithm with remote edge traversal pruning for a Kronecker graph with SCALE29 on a four-socket server (SB4).

Level	Local	Pruned-remote	Remote
0	0	0	4
1	2,319	5	6,715
2	55,174,478	106,233,078	59,254,916
3	3,816,501,320	11,066,187,392	383,443,940
4	373,917,620	1,110,137,452	11,576,835
5	1,137,197	3,109,482	304,082
6	2,854	2,186	6,108
7	4	6	19
Total	4,246,735,792	12,285,669,601	454,592,619
Ratio	25.000%	72.324%	2.676%



(a) UV 2000 with 64 CPU sockets (One rack)



(b) UV 300 with 32 CPU sockets (One rack)

Figure 3: Memory bandwidth GB/s between arbitrary two NUMA nodes.

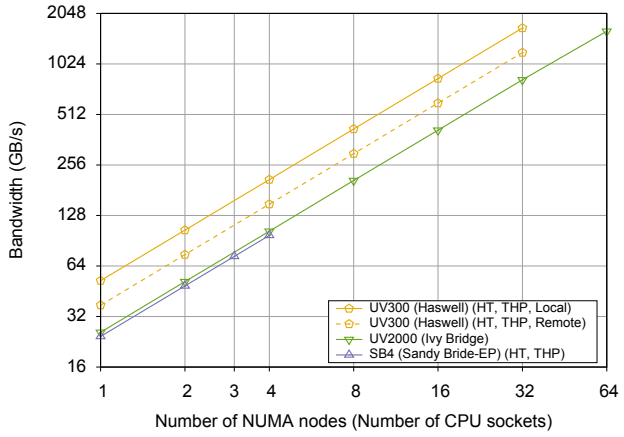


Figure 4: Memory bandwidth (GB/s)

5. NUMERICAL RESULTS

5.1 SGI UV 2000

Fig. 6 shows the weak scaling performances of our previous [22] and current implementation, which collect TEPS scores with fixed problem size as SCALE 26 and SCALE 27 per CPU socket. The previous implementation scaled up to 1,280 threads, and achieves 131 GTEPS for SCALE 32 with 640 threads and 175 GTEPS for SCALE 33 with 1280 threads, respectively. In contrast, the current implementation achieves 152 GTEPS for SCALE 33 with 640 threads – scalability is improved as a result of the pruning of edge traversal for remote memory in the top-down direction. However, we only have results for a maximum of 640 threads. The performance gap between the previous (131 GTEPS for SCALE 33) and current (152 GTEPS for SCALE 34) implementations is 15.8% ($= \frac{152}{131}$) on one rack of UV 2000 with 640 threads.

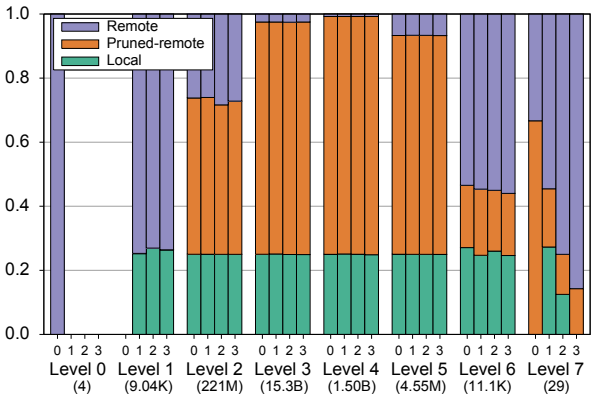


Figure 5: Ratio of traversed edges on a NUMA node in the top-down algorithm with remote edge traversal pruning for a Kronecker graph with SCALE29 on a four-socket server (SB4). Each number in a bracket represents the total number of traversed edges at each level.

Algorithm 3: Top-down with pruning remote traversal

Procedure *NUMA-aware-Top-down*(G, CQ, VS, π)
fork
`/* i-th thread runs on j-th core of k-th CPU */`
1 $(i, j, k) \leftarrow$ ULIBC_get_current_numainfo()
2 $\mathrm{NQ}_k \leftarrow \emptyset$
3 **for** $v \in CQ_k$ **in parallel do**
4 **for** $w \in A_k^B(v)$ **do**
5 **if** $owner(w) = k$ **then**
6 **if** $w \notin VS$ **atomic then**
7 $\pi(w) \leftarrow v$
8 $\mathrm{VS}_k \leftarrow \mathrm{VS}_k \cup \{w\}$
9 $\mathrm{NQ}_k \leftarrow \mathrm{NQ}_k \cup \{w\}$
10 **else**
11 **if** $w \notin F_k$ **atomic then**
12 $F_k \leftarrow F_k \cup \{w\}$
13 $SQ_{\mathrm{owner}(w)} \leftarrow SQ_{\mathrm{owner}(w)} \cup \{(v, w)\}$
14 synchronize
15 **for** $(v, w) \in SQ_k$ **in parallel do**
16 **if** $w \notin VS$ **atomic then**
17 $\pi(w) \leftarrow v$
18 $\mathrm{VS}_k \leftarrow \mathrm{VS}_k \cup \{w\}$
19 $\mathrm{NQ}_k \leftarrow \mathrm{NQ}_k \cup \{w\}$
20 join
21 **return** NQ_k

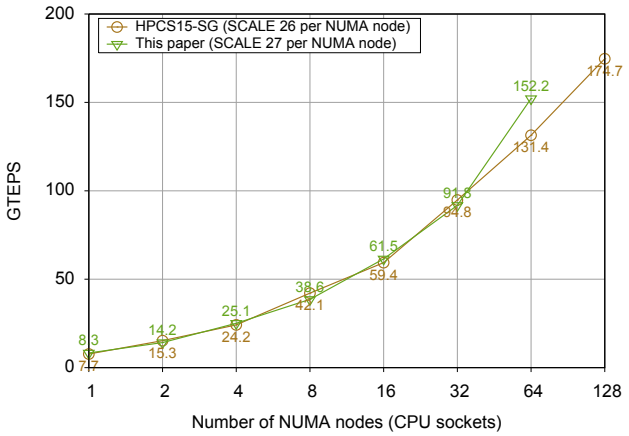


Figure 6: Weak scaling on UV 2000

5.2 SGI UV 300

In this study, we obtained new results on SGI UV 300, which has 32 CPU sockets and 16 TB memory. Fig. 7 depicts TEPS versus number of CPU sockets (NUMA nodes). Table 8 shows the TEPS obtained for 32 CPU sockets. We discuss the results with the following parameters:

- Hyperthreading (HT): {**enabled**, *disabled*}
- Transparent hugepage (THP): {**enabled**, *disabled*}
- Priority mode for memory access: {**local**, *remote*}

For example, "(HT, THP, local)" means Hyperthreading and Transparent hugepage are enabled and priority mode is set for local memory. In this table, a check mark (✓) indicates that a parameter is enabled. First, UV 300 is faster than UV 2000 for large problem sizes. Second, both Hyperthreading (HT) and Transparent hugepage (THP) together improved the performance by 16.49 % ($= \frac{219}{188}$) and 4.78 % ($= \frac{219}{209}$). Third, our implementation applied several techniques that improved the locality of memory access to make it suitable for priority mode set as local memory. Ultimately, the best performance obtained was 219 GTEPS for SCALE 34 with the configuration set as (HT, THP, Local), indicated in bold font above.

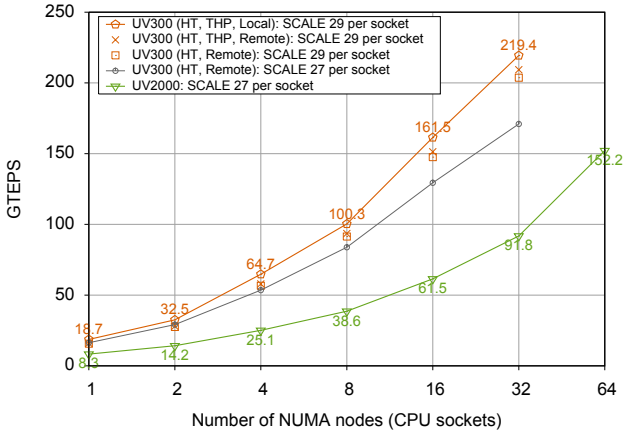


Figure 7: Weak scaling on UV 300

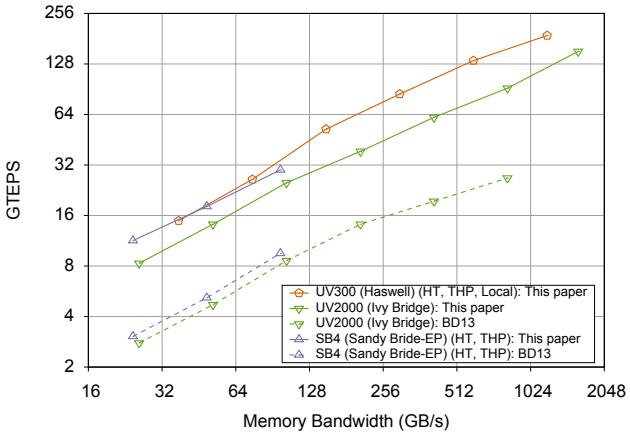
Table 8: GTEPS on UV 300 with 32 CPU sockets

System	SCALE	HT	THP	Mode	GTEPS
UV 2000	32	–	–	–	92
UV 300	32	✓	–	Remote	171
UV 300	34	✓	–	Remote	204
UV 300	34	✓	✓	Remote	209
UV 300	34	*1	✓	Local	188
UV 300	34	✓	✓	Local	219

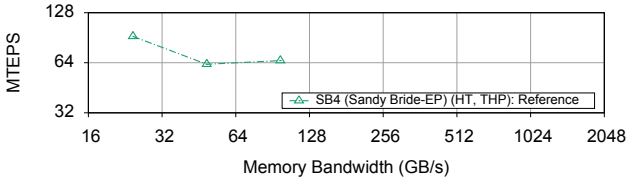
*1 use the number of threads same as physical cores.

5.3 STREAM and Graph500 benchmarks

Finally, the correlativity between the memory bandwidth (bytes per seconds) for the STREAM benchmark and the graph traversal performance (TEPS) is depicted in Fig 8. Each line represents pairs of {Memory bandwidth (in GB/s) of the STREAM benchmark TRIAD operation with 10^7 elements per CPU socket, Graph500 score (GTEPS) with SCALE 27 per CPU socket} for each of 1, 2, 4, 8, 16, and 32 CPU sockets. We obtained the memory bandwidth score via a modified implementation using ULIBC, in which each thread computed the partial TRIAD operation for vectors on local memory only, shown in subsection 3.2. Figure shows correlativity between the memory bandwidth and the graph traversal performance. The optimized Graph500 implementation and our previous implementation are scalable, like the memory bandwidth. In contrast, the reference code of Graph500 is not scalable and cannot exploit the NUMA system efficiently.



(a) GTEPS



(b) MTEPS

Figure 8: TEPS versus Memory bandwidth (GB/s)

6. CONCLUSIONS

In this paper, we presented a new and efficient breadth-first search algorithm for large-scale networks on a single

cache coherent (cc)-NUMA system, in which the pruning technique is adapted for top-down direction of direction-optimizing algorithm. The new algorithm exhibits an improved performance of 152 GTEPS for SCALE 33 from 131 GTEPS for SCALE 32 on UV 2000 with 640 threads. We also evaluated the performance for configurations including Hyperthreading, Transparent hugepage, and priority mode for memory access, and obtained the best performance of 219 GTEPS for SCALE 34 on UV 300. Both the UV 2000 and UV 300 systems are shared-memory supercomputers, but their interconnect topologies differ. Whereas UV 300 connects other compute nodes based on an all-to-all topology via NUMAlink interconnects, UV 2000 connects based on a hypercube topology. As a result, the UV 300 system can achieve a higher performance than the UV 2000 on data intensive computations, such as breadth-first search on the Graph500 benchmark. Furthermore, we focused on the correlation between memory bandwidth and graph traversal performance and confirmed that our implementation is scalable. In future work, we will examine a NUMA-aware graph analysis library for large-scale single-node systems, such as SGI UV systems.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

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

[2] S. Beamer, K. Asanovć, and D.A. Patterson. *Direction-optimizing breadth-first search.* In Proceedings of the ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC12), 2012.

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

[4] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst: *hwloc: A generic framework for managing hardware affinities in HPC applications.* In Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010), 2010.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge MA, 1990.

[6] E. A. Dinic. *Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation.* Soviet Math Doklady, 11:1277–1280, 1970

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

[8] M. Frasca, K. Madduri, and P. Raghavan. *NUMA-aware graph mining techniques for performance and energy efficiency.* In Proceedings of the ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC12), 2012.

[9] 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 Proceedings of the IEEE Int. Symp. Parallel and Distributed Processing (IPDPS 14), 2014.

[10] M. Girvan and M.E.J. Newman. *Community structure in social and biological networks.* In Proceedings Natl. Acad. Sci. USA, 99:7821–826, 2002.

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

[12] L. Huang, H. Jin, L. Yi, and B. Chapman: *Enabling locality-aware computations in OpenMP*, Journal Scientific Programming - Exploring Languages for Expressing Medium to Massive On-Chip Parallelism archive, 18(3–4):169–181, 2010.

[13] *Intel(R) C++ Compiler XE 16.0 User and Reference Guide*, Intel Thread Affinity Interface.

[14] 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.

[15] A. Lenharth and K. Pingali. *Scaling runtimes for irregular algorithms to large-scale NUMA systems.* Computer, IEEE Computer Society, 48(8), 2015.

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

[17] J. Treibig, G. Hager, G. Wellein: *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.* In Proceedings of PSTI 2010, 2010.

[18] K. Ueno and T. Suzumura. *Highly scalable graph search for the Graph500 benchmark.* In Proceedings of 21st Int. ACM Symp. High-Performance Parallel and Distributed Computing (HPDC 12), 2012.

[19] 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.

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

[21] 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, 8488:365–381, 2014.

[22] Y. Yasui and K. Fujisawa: *Fast and scalable NUMA-based thread parallel breadth-first search.* In Proceedings of ACM/IEEE/IFIP Int. Conf. High Performance Computing & Simulation (HPCS 2015), 2015.

[23] K. Zhang, R. Chenm and H. Chen. *NUMA-aware graph-structured analytics.* In Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP715), 2015.