Data Sci. Eng. (2017) 2:22–35
DOI 10.1007/s41019-016-0024-y

Efficient Breadth-First Search on Massively Parallel and Distributed-Memory Machines

Koji Ueno[1] · Toyotaro Suzumura[2] · Naoya Maruyama[3] · Katsuki Fujisawa[4] · Satoshi Matsuoka[5]

Received: 16 October 2016 / Accepted: 13 December 2016 / Published online: 9 January 2017

Abstract There are many large-scale graphs in real world such as Web graphs and social graphs. The interest in large-scale graph analysis is growing in recent years. Breadth-First Search (BFS) is one of the most fundamental graph algorithms used as a component of many graph algorithms. Our new method for distributed parallel BFS can compute BFS for one trillion vertices graph within half a second, using large supercomputers such as the K-Computer. By the use of our proposed algorithm, the K-Computer was ranked 1st in Graph500 using all the 82,944 nodes available on June and November 2015 and June 2016 38,621.4 GTEPS. Based on the hybrid BFS algorithm by Beamer (Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW '13, IEEE Computer Society, Washington, 2013), we devise sets of optimizations for scaling to extreme number of nodes, including a new efficient graph data structure and several optimization techniques such as vertex reordering and load balancing. Our performance evaluation on K-Computer shows that our new BFS is 3.19 times faster on 30,720 nodes than the base version using the previously known best techniques.

✉ Koji Ueno
kojiueno5@gmail.com

[1] Tokyo Institute of Technology, Tokyo, Japan

[2] IBM T.J. Watson Research Center, Westchester County, NY, USA

[3] RIKEN, Kobe, Japan

[4] Kyushu University, Fukuoka, Japan

[5] Tokyo Institute of Technology/AIST, Tokyo, Japan

Keywords Distributed-memory · Breadth-First Search · Graph500

1 Introduction

Graphs have quickly become one of the most important data structures in modern IT, such as in social media where the massive number of users is modeled as vertices and their social connections as edges, and collectively analyzed to implement various advanced services. Another example is to model biophysical structures and phenomena, such as brain's synaptic connections, or interaction network between proteins and enzymes, thereby being able to diagnose diseases in the future. The common properties among such modern applications of graphs are their massive size and complexity, reaching up to billions of edges and trillions of vertices, resulting in not only tremendous storage requirements but also compute power to conduct their analysis.

With such high interest in analytics of large graphs, a new benchmark called the Graph500 [8, 11] was proposed in 2010. Since the predominant use of supercomputers had been for numerical computing, most of the HPC benchmarks such as the Top500 Linpack had been compute centric. The Graph500 benchmark instead measures the data analytics performance of supercomputers, in particular those for graphs, with the metric called traversed edges per second or TEPS. More specifically, the benchmark measures the performance of Breadth-First Search (BFS), which is utilized as a kernel for important and more complex algorithms such as connected components analysis and centrality analysis. Also, the target graph used in the benchmark is a scale-free, small-diameter graph called the Kronecker graph, which is known to model realistic

graphs arising out of practical applications, such as Web and social networks, as well as those that arise from life science applications. As such, attaining high performance on the Graph500 represents the important abilities of a machine to process real-life, large-scale graphs arising from big-data applications.

We have conducted a series of work [11–13] to accelerate BFS in a distributed-memory environment. Our new work extends the data structures and algorithm called hybrid BFS [2] that is known to be effective small-diameter graphs, so that it scales to top-tier supercomputers with tens of thousands of nodes with million-scale CPU cores with multi-gigabyte/s interconnect. In particular, we apply our algorithm to the Riken's K-Computer [15] with 82,944 compute nodes and 663,552 CPU cores, once the fastest supercomputer in the world on the Top500 in 2011 with over 10 Petaflops. The result obtained is currently No. 1 on the Graph500 for two consecutive editions in 2016, with significant TEPS performance advantage compared to the result obtained on the Sequoia supercomputer hosted by Lawrence Livermore National Laboratory in the USA, which is a machine with twice the size and performance compared to the K-Computer, with over 20 Petaflops and embodying approximately 1.6 million cores. This demonstrates that top supercomputers compete for the top ranks on the Graph500, but the Top500 ranking does not necessarily directly translate in this regard; rather architectural properties other than the amount of FPUs, as well as algorithmic advances, play a major role in attaining top performance, indicating the importance of codesign of future top-level machines including those for exascale, with graph-centric applications in mind .

In fact, the top ranks of the Graph500 has been historically dominated by large-scale supercomputers to date, with other competing infrastructures such as Clouds being notably missing; performance measurements of the various work including ours reveal that this is fundamental, in that interconnect performance plays a significant role in the overall performance of large-scale BFS, and this is one of the biggest differentiators between supercomputers and Clouds.

2 Background: Hybrid BFS

2.1 The Base Hybrid BFS Algorithm

We first describe the background BFS algorithms, including hybrid algorithm as proposed in [2]. Figure 1 shows the standard sequential textbook BFS algorithm. Starting from the source vertex, the algorithm conducts the search by effectively expanding the *"frontier"* set of vertices in a breadth-first manner from the root. We refer to this search direction as *"top-down."*

```
1: function BREADTH-FIRST-SEARCH(vertices, source)
2:     frontier ← {source}
3:     next ← {}
4:     parents ← [−1, −1, ···, −1]
5:     while frontier ≠ {} do
6:         top-down-step (vertices, frontier, next, parents)
7:         frontier ← next
8:         next ← {}
9:     end while
10:    return parents
11: end function
12: function TOP-DOWN-STEP(vertices, frontier, next, parents)
13:    for v ∈ frontier do
14:        for n ∈ neighbors[v] do
15:            if parents[n] = -1 then
16:                parents[n] ← v
17:                next ← next ∪ {n}
18:            end if
19:        end for
20:    end for
21: end function
```

Fig. 1 Top-down BFS

A contrasting approach is *"bottom-up"* BFS as shown in Fig. 2. This approach is to start from the vertices that have not been visited and iterate with each step investigating whether a frontier node is included in its direct neighbor. If it is, then the node is added to the frontier of visited nodes for the next iteration. In general, this *"bottom-up"* approach is more advantageous over top-down when the frontier is large, as it will quickly identify and mark many nodes as visited. On the other hand, top-down is advantageous when the frontier is small, as bottom-up will result in wasteful scanning of many unvisited vertices and their edges without much benefit.

For a large but small-diameter graphs such as the Kronecker graph used in the Graph500, the *hybrid BFS algorithm* [2] (Fig. 3) that heuristically minimizes the number of edges to be scanned by switching between top-down and bottom-up, has been identified as very effective in significantly increasing the performance of BFS.

```
1: function BOTTOM-UP-STEP(vertices, frontier, next, parents)
2:     for v ∈ vertices do
3:         if parents[v] = -1 then
4:             for n ∈ neighbors[v] do
5:                 if n ∈ frontier then
6:                     parents[v] ← n
7:                     next ← next ∪ {v}
8:                     break
9:                 end if
10:            end for
11:        end if
12:    end for
13: end function
```

Fig. 2 A step in bottom-up BFS

```
1: function HYBRID-BFS(vertices, source)
2:     frontier ← {source}
3:     next ← {}
4:     parents ← [-1,-1,···,-1]
5:     while frontier ≠ {} do
6:         if next-direction() = top-down then
7:             top-down-step (vertices, frontier, next, parents)
8:         else
9:             bottom-up-step (vertices, frontier, next, parents)
10:        end if
11:        frontier ← next
12:        next ← {}
13:    end while
14:    return parents
15: end function
```

Fig. 3 Hybrid BFS

2.2 Parallel and Distributed BFS Algorithm

In order to parallelize the BFS algorithm over distributed-memory machines, it is necessary to spatially partition the graphs. A proposal by Beamer et. al. [3] conducts 2-D partitioning of the adjacency matrix of the graph in two dimensions, as shown in Fig. 4, where adjacency matrix A is partitioned into $R \times C$ submatrices.

Each of the submatrices is assigned to a compute node; the compute nodes themselves are virtually arranged into a $R \times C$ mesh, being assigned a 2-D index $P(i, j)$. Figures 5 and 6 illustrate the top-down and bottom-up parallel-distributed algorithms with such a partitioning scheme. In the figures, $P(:, j)$ means all the processors in j-th column of 2-D processor mesh, and $P(i, :)$ means all the processors in i-th row of 2-D processor mesh. Line 8 of Fig. 5 performs the allgatherv communication operation among all the processors in j-th column, and line 15 performs the alltoallv communication operation among all the processors in i-th row.

In Figs. 5 and 6, f, n, and π correspond to frontier, next, and parent in the base sequential algorithms, respectively. Allgatherv() and alltoallv() are standard MPI collectives. Beamer [3]'s proposal encodes f, c, n, w as 1 bit per vertex for optimization. Parallel-distributed hybrid BFS is similar to the sequential algorithm in Fig. 4, heuristically switching between top-down and bottom-up per each iteration step, being essentially a hybrid of algorithms in Figs. 5 and 6.

In parallel 2-D bottom-up BFS algorithm in Fig. 6, each search step is broken down into C substeps assuming that

$$A = \begin{pmatrix} A_{1,1} & \cdots & A_{1,C} \\ \vdots & \ddots & \vdots \\ A_{R,1} & \cdots & A_{R,C} \end{pmatrix}$$

Fig. 4 $R \times C$ partitioning of adjacency matrix A

```
1: function PARALLEL-2D-TOP-DOWN(A, source)
2:     f ← {source}
3:     n ← {}
4:     π ← [−1, −1, ···, −1]
5:     for all compute nodes P(i, j) in parallel do
6:         while f ≠ {} do
7:             transpose-vector(f_{i,j})
8:             f_i = allgatherv(f_{i,j},P(:, j))
9:             t_{i,j} ← {}
10:            for u ∈ f_i do
11:                for v ∈ A_{i,j}(:, u) do
12:                    t_{i,j} ← t_{i,j} ∪ {(u, v)}
13:                end for
14:            end for
15:            w_{i,j} ← alltoallv(t_{i,j}, P(i, :))
16:            for (u, v) ∈ w_{i,j} do
17:                if π_{i,j}(v) = −1 then
18:                    π_{i,j}(v) ← u
19:                    n_{i,j} ← n_{i,j} ∪ v
20:                end if
21:            end for
22:            f ← n
23:            n ← {}
24:        end while
25:    end for
26:    return π
27: end function
```

Fig. 5 Parallel-distributed 2-D top-down algorithm

an adjacency matrix is partitioned into $R \times C$ submatrices in a two-dimensional, and during each substep, a given vertex's edges will be examined by only one processor. During each substep, a processor processes $1/C$ of the assigned vertices in the processor row. After each substep, it passes on the responsibility for those vertices to the processor to its right and accepts new vertices from the processor to its left. This pairwise communication sends which vertices have been completed (called found parents), so that the next processor will have the knowledge to skip examining over them. This has the effect of the processor responsible for processing a vertex rotating right along the row for each substep. When a vertex finds a valid parent to become visited, its index along with its discovered parent is queued up and sent to the processor responsible for the corresponding segment of the parent array to update it. Each step of the algorithm in Fig. 6 has four major operations [3];

Frontier Gather (per step) (lines 8–9)

Each processor is given the segment of the frontier corresponding to their assigned submatrix.

Local discovery (per substep) (lines 11–20)

Search for parents with the information available locally.

Parent Updates (per substep) (lines 21–25)

Send updates of children that found parents and process updates for own segment of parents.

Fig. 6 Parallel-distributed 2-D bottom-up algorithm

```
1: function PARALLEL-2D-BOTTOM-UP(A, source)
2:     f ← {source}                                   ▷ bitmap for frontier
3:     c ← {source}                                   ▷ bitmap for completed
4:     n ← {}
5:     π ← [−1, −1, ···, −1]
6:     for all compute nodes P(i, j) in parallel do
7:         while f ≠ {} do
8:             transpose-vector(f_{i,j})
9:             f_i =allgatherv(f_{i,j}, P(:, j))
10:            for s in 0 ... C − 1 do                ▷ C sub-steps
11:                t_{i,j} ← {}
12:                for u ∈ c_{i,j} do
13:                    for v ∈ A_{i,j}(u, :) do
14:                        if v ∈ f_i then
15:                            t_{i,j} ← t_{i,j} ∪ {(v, u)}
16:                            c_{i,j} ← c_{i,j}\u
17:                            break
18:                        end if
19:                    end for
20:                end for
21:                w_{i,j} ← sendrecv(t_{i,j}, P(i, j + s), P(i, j − s))
22:                for (v, u) ∈ w_{i,j} do
23:                    π_{i,j}(v) ← u
24:                    n_{i,j} ← n_{i,j} ∪ v
25:                end for
26:                c_{i,j} ← sendrecv(c_{i,j}, P(i, j + 1), P(i, j − 1))
27:            end for
28:            f ← n
29:            b ← {}
30:        end while
31:    end for
32:    return π
33: end function
```

Rotate Completed (per substep) (line 26)

Send completed to the right neighbor and receive completed for the next substep from the left neighbor.

3 Problems of Hybrid BFS in Extreme-Scale Supercomputers

Although the algorithm in Sect. 2 would work efficiently on a small-scale machine, for extremely large, up to and beyond million-core scale supercomputers toward exascale, various problems would manifest themselves which severely limit the performance and scalability of BFS. We describe the problems in Sect. 3 and present our solutions in Sect. 4.

3.1 Problems with the Data Structure of the Adjacency Matrix

The data structure describing the adjacency matrix is of significant importance as it directly affects the computational complexity of graph traversal. For small machines, the typical strategy is to employ the Compressed Sparse Row (CSR) format, commonly employed in numerical computing to express sparse matrices. However, we first show that direct use of CSR is impractical due to its memory requirements on a large machine; we then show that the existing proposed solutions, DCSR [4] and Coarse index + Skip list [6] that intend to reduce the footprint at the cost of increased computational complexity, are still insufficient for large graphs with significant computational requirement.

3.1.1 Compressed Sparse Row (CSR)

CSR utilizes two arrays, *dst* that holds the destination vertex ID of the edges in the graph and *row-starts* that describes the offset index of the edges of each vertex in the *dst* array. Given a graph with V vertices and E edges, the size of $dst = E$ and $row\text{-}starts = V$, respectively, so the required memory would be as follows in a sequential implementation:

$$V + E \tag{1}$$

For parallel-distributed implementation with $R \times C$ partitioning, if we assume that the edges and vertices are distributed evenly, since the number of rows in the distributed submatrices is V / R, the required memory per node is:

$$\frac{V}{R}+\frac{E}{RC} \tag{2}$$

By denoting the average vertices per node as V' and the average degree of the graph as $\hat{d}$, the following equation holds:

$$V'=\frac{V}{RC}, E=V\hat{d} \tag{3}$$

(2) can then be expressed as follows:

$$V'C+V'\hat{d} \tag{4}$$

This indicates that, for large machines, as C gets larger, the memory requirement per node increases, as the memory requirement of row-starts is $V'C$. In fact, for very large graphs on machines with thousands of nodes, row-starts can become significantly larger than *dst*, making its straightforward implementation impractical.

There is a set of work that proposes to compress row-starts, such as DCSR [4] and *Coarse index + Skip list* [6], but they involve non-negligible performance overhead as we describe below:

3.1.2 DCSR

DCSR [4] was proposed to improve the efficiency of matrix-matrix multiplication in a distributed-memory environment. The key idea is to eliminate the row-starts value for rows that has no nonzero values, thereby compressing row-starts. Instead, two supplemental data structures called the JC and AUX arrays are employed to calculate the appropriate offset in the *dst* array. The drawback is that one needs to iterate in order to navigate over the JC array from the AUX array, resulting in significant overhead for repeated access of sparse structures, which is a common operation for BFS.

3.1.3 Coarse Index + Skip List

Another proposal [6] was made in order to efficiently implement Breadth-First Search for 1-D partioning in a distributed-memory environment. Sixty-four rows of nonzero elements are batched into a *skip list*, and by having the row-starts hold the pointer to the *skip list*, this method compresses the overall size of the row-starts to be 1/64th the original size. Since each skip list embodies 64 rows of data, we can traverse all 64 rows contiguously, making algorithms with batched row access efficient in addition to data compression. However, for sparse accesses, on average one would have to traverse and skip over 31 elements to access the designated matrix element, potentially introducing significant overhead.

3.1.4 Other Sparse Matrix Formats

There are other known sparse matrix formats that do not utilize row-starts [9], significantly saving memory; however, although such formats would be useful for algorithms that systematically iterate over all elements of a matrix, they perform badly for BFS where individual accesses to the edges of a given vertex need to be efficient.

3.2 Problems with Communication Overhead

Hybrid BFS with 2-D partitioning scales for small number of nodes, but its scalability is known to quickly saturate when the number of nodes scales beyond thousands [3].

In particular, for distributed hybrid BFS over small-diameter large graph such as the Graph500 Kronecker graph, it has been reported that bottom-up search involves significant longer execution time compared to top-down [3, 14]. Table 1 shows the communication cost of bottom-up search when f, c, n, w are implemented as bitmaps as proposed in [3]. Each operation corresponds to the program in the following fashion: Transpose is line 8 of Fig. 6, Frontier Gather is line 9, Parent Update is line 21, and Rotate Completed is line 26, respectively.

As we can see in Table 1, the communication cost of Frontier Gather and Rotate Completed is proportional to R and C in the submatrix portioning—being one of the primary sources overhead when number of nodes are in the thousands or more. Moreover, lines 21 and 26 involve synchronous communication with other nodes, and the number of communication is proportional to C, again becoming significant overhead. Finally, it is very difficult to achieve perfect load balancing, as a small number of vertices tend to involve number of edges that could be orders of magnitude larger than the average; this could result in sever load imbalance in simple algorithms that assume even distribution of vertices and edges.

Such difficulties have been the primary reasons why one could not obtain near linear speedups, even in weak scaling, as the number of compute nodes the associated graph sizes increased to thousands or more on a very large machine. We next introduce our extremely scalable hybrid BFS that alleviates these problems, to achieve utmost scalability for Graph500 execution on the K-Computer.

4 Our Extremely Scalable Hybrid BFS

The problems associated with previous algorithms are largely storage and communication overheads of extremely large graphs scaling to be analyzed over thousands of

Table 1 Communication cost of bottom-up search [3]

Operation	Comm type	Comm complexity per step	Data transfer per each search (64 bit word)
Transpose	P2P	$O(1)$	$s_b V/64$
Frontier Gather	Allgather	$O(1)$	$s_b VR/64$
Parent Updates	P2P	$O(C)$	$2V$
Rotate Completed	P2P	$O(C)$	$s_b VC/64$

nodes or more. These are fundamental to the fact that we are handling irregular, large-scale "big" data structures and not floating point numerical values. In order to alleviate the problems, we propose several solutions that are unique to graph algorithms

4.1 Bitmap-Based Sparse Matrix Representation

First, our proposed bitmap-based sparse matrix representation allows extremely compact representation of the adjacency matrix, while still being very efficient in retrieving the edges between given vertices. We compress the CSR row-starts data structure by only holding the starting position of the sequence of edges for vertices that has one or more edges and then having an additional bitmap to identify whether a given vertex has more than one edge or not, one bit per vertex.

In our bitmap-based representation, since the sequence of edges is held in row-starts in the same manner as CSR, the main point of the algorithm is to how to identify the starting index of the edges given a vertex efficiently, as shown in Fig. 7. Here, B is number of bits in a word (typically 64), "$\ll$" and "&" are the bit-shift and bitwise operators, and mod is the modulo operator. Given a vertex v, the index position of v in the row-starts corresponds to the number of vertices with nonzero edges from the vertex zero, which is equivalent to the number of bits that are 1 leading up to the v'th position in the bitmap. We further optimize this calculation by counting the summation of the number of 1 bits on a word-by-word basis and store it in the *offset* array. This effectively allows constant calculation of the number of nonzero bits for v by looking at the offset value and the number of bits that are one leading up to the v's position in that particular word.

Table 2 shows a comparative example of bitmap-based sparse matrix representation with 8 vertices and 4 edges. As we observe, much of the repetitive waste resulting from relatively small number of edges compared to vertices arising in CSR is minimized. Table 3 shows the actual savings we achieve over CSR in a real setting in a Graph500 benchmark. Here, we partition a graph with 16 billion vertices and 256 billion edges into $64 \times 32 = 2048$ nodes in 2-D. Here, we achieve similar level of compression as previous work such as DCSR and Coarse index + Skip list, achieving nearly 60% reduction in space. As we see later, this compression is achieved with minimal execution overhead, in contrast to the previous proposals.

4.2 Reordering of the Vertex IDs

Another associated problem with BFS is the randomness of memory accesses of graph data, in contrast to traditional numerical computing using CSR such as the Conjugate Gradient method, where the access to the row elements of a matrix can become contiguous. Here, we attempt to exploit similar locality properties.

The basic idea is as follows: As described in Sect. 2.2, much of the information regarding hybrid BFS is held in bitmaps that represent the vertices, each bit corresponding to a vertex. When we execute BFS over a graph, higher-

Fig. 7 Bitmap-based sparse matrix: algorithms to calculate the offset and identify the start and end indices of a row of edges given a vertex

```
1: function MAKE-OFFSET(offset, bitmap)
2:     i ← 0
3:     offset[0] ← 0
4:     for each word w of bitmap do
5:         offset[i + 1] ← offset[i] + popcount(w)
6:         i ← i + 1
7:     end for
8: end function
9: function ROW-START-END(offset, bitmap, row-starts, v)
10:    w ← v/B
11:    b ← (1 ≪ (v mod B))
12:    if (bitmap[w] & b) ≠ 0 then
13:        p ← offset[w] + popcount(bitmap[w] & (b − 1))
14:        return (row-starts[p], row-starts[p + 1])
15:    end if
16:    return (0, 0)                          ▷ Vertex v has no edge
17: end function
```

Table 2 Examples of bitmap-based sparse matrix representation

Edges list	SRC	0 0 6 7
	DST	4 5 3 1
CSR	Row-starts	0 2 2 2 2 2 2 3 4
	DST	4 5 3 1
Bitmap-based sparse matrix representation	Offset	0 1 3
	Bitmap	1 0 0 0 0 0 1 1
	Row-starts	0 2 3 4
	DST	4 5 3 1
DCSR	AUX	0 1 1 3
	JC	0 6 7
	Row-starts	0 2 3 4
	DST	4 5 3 1

Table 3 Theoretical order and the actual per-node measured memory consumptions of bitmap-based CSR compared to previous proposals

Data structure	CSR		Bitmap-based CSR	
	Order	Actual	Order	Actual
Offset	–	–	$V'C/64$	32 MB
Bitmap	–	–	$V'C/64$	32 MB
Row-starts	$V'C$	2048 MB	$V'p$	190 MB
DST	$V'\hat{d}$	1020 MB	$V'\hat{d}$	1020 MB
Total	$V'(C+\hat{d})$	3068 MB	$V'(C/32+p+\hat{d})$	1274 MB
Data structure	DCSR		Coarse index + Skip list	
	Order	Actual	Order	Actual
AUX	$V'p$	190 MB	–	–
JC	$V'p$	190 MB	–	–
Row-starts	$V'p$	190 MB	$V'C/64$	32 MB
DST or skip list	$V'\hat{d}$	1020 MB	$V'\hat{d}+V'p$	1210 MB
Total	$V'(3p+\hat{d})$	1590 MB	$V'(C/64+p+\hat{d})$	1242 MB

We partition a Graph500 graph with 16 billion vertices and 256 billion edges into $64 \times 32 = 2048$ Nodes

degree vertices are typically accessed more often; as such, by clustering access to such vertices by reordering them according to their degrees (i.e., # of edges), we can expect to achieve higher locality. This is similar to switching rows in a matrix in a sparse numerical algorithm to achieve higher locality. In [12], they proposed such reordering for top-down BFS, where they only utilize the reordered vertices where needed, while maintaining the original BFS tree with original vertex IDs for overall efficiency. Unfortunately, this method cannot be used for hybrid BFS; instead, we propose the following algorithm.

Reordered IDs of the vertices are computed by sorting them top-down according to their degrees on a per-node basis and then reassigning the new IDs according to their order. We do not conduct any inter-node reordering. A subadjacency matrix on each node stores reordered IDs of the vertices. The mapping information between original vertex ID and its reordered vertex ID is maintained by an owner node where the vertex is located. When constructing an adjacency matrix of the graph, the original vertex ID is converted to the reordered ID by (a) firstly performing all-to-all communication once over all the nodes in a row of processor grid in 2-D partitioning to compute the degree information of each vertex, and then (b) secondly computing the reordered IDs by sorting all the vertices according to their degrees and then (c) thirdly performing all-to-all communication again over all the nodes in a column and a row of processor grid in order to convert the vertex IDs in the subadjacency matrix on each node to the reordered IDs.

The drawback with this scheme requires expensive all-to-all communication multiple times: Since the resulting BFS tree had the reordered IDs for the vertices, we must reassign their original IDs. However, if we are to conduct

Table 4 Adding the original IDs for both the source and the destination

Offset	0 1 3
Bitmap	1 0 0 0 0 0 1 1
SRC(Orig)	2 0 1
Row-starts	0 2 3 4
DST	2 3 0 1
DST(Orig)	4 5 3 1

such reassignment at the very end, the information must be exchanged among all the nodes using a very expensive all-to-all communication for large machines again, since the only node that has the original ID info of each vertex is the node that owns it. In fact, we show in Sect. 6 that all-to-all is a significant impediment in our benchmarks.

The solution to this problem is to add two arrays SRC(Orig) and DST(Orig) as shown in Table 4. Both arrays hold the original indices of the reordered vertices. When the algorithm writes to the resulting BFS tree, the original ID is referenced from either of the arrays instead of the reordered ID, avoiding all-to-all communication. Also, a favorable by-product of vertex reordering is removal of vertices with no edges, allowing further compaction of the data structure, since such vertices will never show up in the resulting BFS tree.

4.3 Optimizing Inter-Node Communications for Bottom-Up BFS

The original bottom-up BFS algorithm shown in Fig. 6 conducts communication per each substep, C times per each iteration assuming that we have 2-D partitioning of $R \times C$ for an adjacency matrix). For large systems, such frequent communication presents significant overhead and thus subject to the following optimizations:

4.3.1 Optimizing Parent Updates Communication

Firstly, we cluster the Parent Updates communication. The sendrecv() communication for line 21 in Fig. 6 sends a request called *"Parent Updates"* to update the BFS tree located at the owner node with the vertices that found a parent in the BFS tree, but such a request can be sent at any time, even after other processing is finished. As such we cluster the Parent Updates as Alltoallv() communication as shown in Fig. 8.

4.3.2 Overlapping Computation and Communication in Rotate Completed Operation

We also attempt to overlap computation in lines 12–20 (Fig. 8) and communication in line 21 in *"Rotate Completed"* operation mentioned in Sect. 2. If the substeps are set to C steps as the original method, we would not be able to overlap the computation and communication since the computational result depends on the result in a previous substep. Thus, we increase this substep from C to multiple substeps such as 2C, 4C. For the K-Computer described in Sect. 5, we increase this to 4C. If the substeps is set to 4C, the computational result depends on the one in 4 substeps before, and we can perform parallel execution by overlapping computation and communication for 4 substeps. In this case, when the computation is performed for 2 substeps, the communication for other 2 substeps are simultaneously executed. The communication is accelerated by allocating these 2 substeps to 2 different communication channels in the 6-D torus network of the K-Computer that supports multiple channel communication using rDMA.

4.4 Reducing Communication with Better Partitioning

We further reduce communication via better partitioning of the graph. The original simple 2-D partitioning by Beamer [3] requires transpose-vector communication per each step. Yoo [16] improves on this by employing block-cyclic distribution, eliminating the need for transpose vector at the cost of added code complexity. We adapt Yoo's method so that it becomes applicable to hybrid BFS (Fig. 9; Table 5).

4.5 Load Balancing the Top-Down Algorithm

We resolve the following load-balancing problem for the top-down algorithm. As shown in Fig. 5 lines 10–14, we need to create $t_{i,j}$ from the edges of each vertex in the frontier; this is implemented so that the each vertex pair of the edges is placed in a temporary buffer and then copied to the communication buffer just prior to alltoallv(). Here, as we see in Fig. 10, thread parallelism is utilized so that each thread gets assigned equal number of frontier vertices. However, since the distribution of edges per each vertex is quite uneven, this will cause significant load imbalance among the threads.

A solution to this problem is shown in Fig. 11, where we conduct partitioning and thread assignment per destination nodes. We first extract the range of edges and copy the edges directly without copying into a temporary buffer. In the figure, owner(v) is a function that returns the owner node of vertex v and edge-range$(A_{i,j}(:,u),k)$ returns the range in edge list $A_{i,j}(:,u)$ for a given owner node k using binary search, as the edge list is sorted in destination ID order. One caveat, however, is when the vertex has only a small number of edges; in such a case, the edge-range data $r_{i,j,k}$ could become larger and thus inefficient. We alleviate

Fig. 8 Bottom-up BFS with optimized inter-node communication

```
1: function PARALLEL-2D-BOTTOM-UP-OPT(A, source)
2:     f ← {source}                                   ▷ bitmap for frontiers
3:     c ← {source}                                   ▷ bitmap for completed
4:     n ← {}
5:     π ← [−1, −1, ···, −1]
6:     for all compute nodes P(i, j) in parallel do
7:         while f ≠ {} do
8:             transpose-vector(f_{i,j})
9:             f_i =allgatherv(f_{i,j}, P(:, j))
10:            for s in 0 ... C − 1 do
11:                t_{i,j} ← {}
12:                for u ∈ c_{i,j} do
13:                    for v ∈ A_{i,j}(u, :) do
14:                        if v ∈ f_i then
15:                            t_{i,j} ← t_{i,j} ∪ {(v, u)}
16:                            c_{i,j} ← c_{i,j}\u
17:                            break
18:                        end if
19:                    end for
20:                end for
21:                c_{i,j} ←sendrecv(c_{i,j}, P(i, j + 1), P(i, j − 1))
22:            end for
23:            w_{i,j} ←alltoallv(t_{i,j}, P(i, :))
24:            for (v, u) ∈ w_{i,j} do
25:                π_{i,j}(v) ← u
26:                n_{i,j} ← n_{i,j} ∪ v
27:            end for
28:            f ← n
29:            n ← {}
30:        end while
31:    end for
32:    return π
33: end function
```

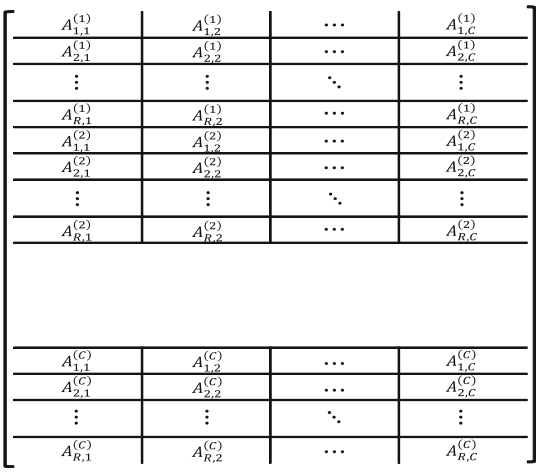

Fig. 9 Block-cyclic 2-D distribution proposed by Yoo [16]

```
1: function TOP-DOWN-SENDER-NAIVE(A_{i,j}, f_i)
2:     for u ∈ f_i in parallel do
3:         for v ∈ A_{i,j}(:, u) do
4:             k ← owner(v)
5:             t_{i,j,k} ← t_{i,j,k} ∪ {(u, v)}
6:         end for
7:     end for
8: end function
```

Fig. 10 Simple thread parallelism for top-down BFS

this problem by using a hybrid method depending on the number of edges, where we switch between the simple copy method and the range method according to the number of edges.

Table 5 Bitmap-based CSR data communication volume (difference from Table 1 is in italics)

Operation	Comm type	Comm complexity per step	Data transfer per each search (64 bit word)
Frontier Gather	Allgather	$O(1)$	$s_b VR/64$
Parent Updates	*Alltoall*	$O(1)$	$2V$
Rotate Completed	P2P	$O(C)$	$s_b VC/64$

```
1: function TOP-DOWN-SENDER-LOAD-BALANCED(A_{i,j}, f_i)
2:     for u ∈ f_i in parallel do
3:         for k ∈ P(i,:) do
4:             (v_0, v_1) ← edge-range(A_{i,j}(:, u), k)
5:             r_{i,j,k} ← r_{i,j,k} ∪ {(u, v_0, v_1)}
6:         end for
7:     end for
8:     for k ∈ P(i,:) in parallel do
9:         for (u, v_0, v_1) ∈ r_{i,j,k} do
10:            for v ∈ A_{i,j}(v_0 : v_1, u) do
11:                t_{i,j,k} ← t_{i,j,k} ∪ {(u, v)}
12:            end for
13:        end for
14:    end for
15: end function
```

Fig. 11 Load-balanced thread parallelism for top-down BFS

5 Machine Architecture-Specific Communication Optimizations for the K-Computer

The optimizations we have proposed so far are applicable to any large supercomputer that supports MPI+OpenMP hybrid parallelism. We now present further optimizations specific to the K-Computer, exploiting its unique architectural capabilities. In particular, the node-to-node interconnect employed in the K-Computer is a proprietary *"Tofu"* network that implements a six-dimensional torus topology, with high injection bandwidth and multi-directional DMA to achieve extremely high performance in communication-intensive HPC applications. We exploit the features of the Tofu network to achieve high performance on BFS as well.

5.1 Mapping to the Six-Dimensional Torus *"Tofu"* Network

Since our bitmap-based hybrid BFS employs two-dimensional $R \times C$ partitioning, there is a choice of how to map this onto the six-dimensional Tofu network, whose dimensions are named "x, y, z, a, b, c." One obvious choice is to assign three dimensions to each R and C (say $R = x, y, z$ and $C = a, b, c$), allowing physically proximal communications for adjacent nodes in the $R \times C$ partitioning. Another interesting option is to assign $R = y, z$ and $C = x, a, b, c$, where we achieve square 288×288 partitioning when we use the entire K-Computer. We test both cases in the benchmark for comparison.

```
1: function SENDRECV-COMPLETED(c_{i,j}, s)
2:     route ← s mod 2
3:     if route = 0 then
4:         c_{i,j} ← sendrecv(c_{i,j}, P(i, j+1), P(i, j−1))
5:     else
6:         c_{i,j} ← sendrecv(c_{i,j}, P(i, j−1), P(i, j+1))
7:     end if
8: end function
```

Fig. 12 Bidirectional communication in bottom-up BFS

5.2 Bidirectional Simultaneous Communication for Bottom-Up BFS

Each node on the K-Computer has six 5 Gigabyte/s bidirectional links to comprise a six-dimensional torus and allows simultaneous DMA to four of the six links. BlueGene/Q has a similar mechanism. By exploiting such simultaneous communication capabilities over multiple links, we can significantly speed up the communication for bottom-up BFS. In particular, we have optimized Rotate Completed communication by communicating simultaneously to both directions, as shown in Fig. 12. Here, ci,j is the data to be communicated, and s is the number of steps up to 2C or 4C steps. We case-analyze s to even/odd to communicate to different directions simultaneously .

One thing to note is that, despite these K-Computer-specific optimizations, we still solely use the vendor MPI for communication and do not employ any machine-specific low-level communication primitives that are non-portable.

6 Performance Evaluation

We now present the results of the Graph500 benchmark using our hybrid BFS on the entire K-Computer. The Graph500 benchmark measures the performance of each machine by the (traversed edges per second (TEPS) value of the BFS algorithm on a synthetically generated Kronecker graphs, with parameters A=0.57, B=0.19, C=0.19, D=0.05. The size of the graph is expressed by the scale parameter where the $\#\text{vertices} = 2^{Scale}$, and the $\#\,\text{edges} = \#\,\text{vertices} \times 16$.

The K-Computer is located at the Riken AICS facility in Japan, with each node embodying a 8-core Fujitsu SPARC64 VIIIfx processor and 16 GB of memory. The Tofu network composes a six-dimensional torus as mentioned, with each link being bidirectional 5GB/s. The total number of nodes is 82,944, or embodying 663,552 CPU cores and approximately 1.3 Petabytes of memory.

6.1 Effectiveness of the Proposed Methods

We measure the effectiveness of the proposed methods using up to 15,360 nodes of the K-Computer. We increased the number of nodes in the increments of 60, with minimum being Scale 29 (approximately 537 million vertices and 8.59 billion edges), up to Scale 37. We picked a random vertex as the root of BFS and executed each

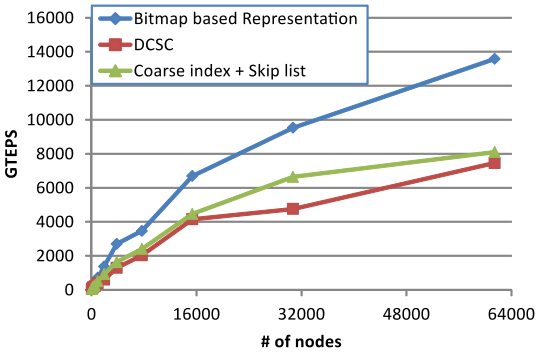

Fig. 13 Evaluation of bitmap-based sparse matrix representation compared to previously proposed methods (K-Computer, weak scaling)

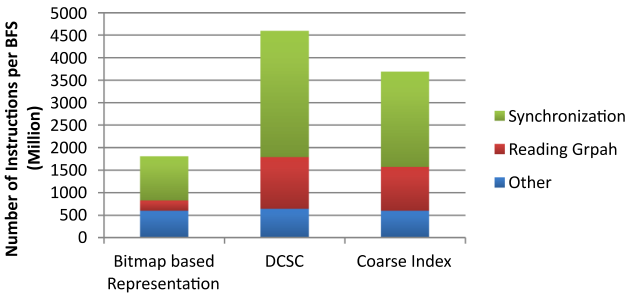

Fig. 14 Performance breakdown—# of instructions per step (Scale 33 graph on 1008 nodes)

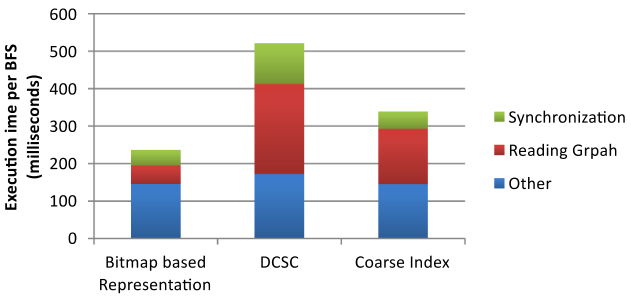

Fig. 15 Performance breakdown—execution time per step (Scale 33 graph on 1008 nodes)

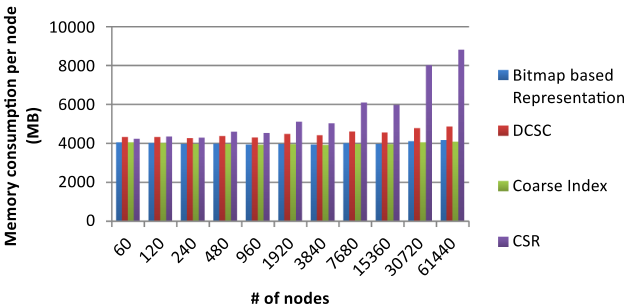

Fig. 16 Memory consumption per node on BFS execution

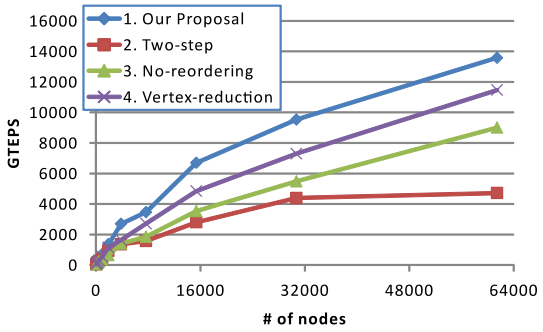

Fig. 17 Reordering of vertex IDs and comparisons to other proposed methods

benchmark 300 times. The reported value is the median of the 300 runs.

We first compared our bitmap-based sparse matrix representation to previous approaches, namely DCSR [4] and Coarse index + Skip list [6]. Figure 13 shows the weak scaling result of the execution performance in GTEPS, and Figs. 14, 15, and 16 shows various execution metrics—#instructions, time, and memory consumed. The processing of *"Reading Graph"* in Figs. 14 and 15 corresponds to lines 10–14 of Fig. 5 and lines 12–20 of Fig. 6. *"Synchronization"* is the inter-thread barrier synchronization over all computation. Since the barrier is implemented with *"spin wait,"* the number of executed instructions for this barrier is large compared with others.

Our proposed method excels in all aspects in comparison with others in performance, while being modest in memory consumption. In particular, for graph reading and manipulation, our proposed method is 5.5 times faster than DCSR and 3.0 times faster than Coarse index + Skip list, while the memory consumption is largely equivalent.

Figure 17 shows the effectiveness of reordering of vertex ID. We compare the four methodological variations, namely (1) our proposed method, (2) reorder but reassign the original ID at the very end using alltoall(), (3) no vertex reordering, and (4) no vertex reordering but pre-eliminate the vertices with no edges. The last method (4) was introduced to assess the effectiveness of our approach more purely with respect to locality improvement, as (1) embodies the effect of both locality improvement and zero-edge vertex elimination. Figure 17 shows that method (2) involves significant overhead in alltoall() communication for large systems, even trailing the non-reordered case. Method (4) shows good speedup over (3), and this is due to the fact that the Graph500 graphs generated at large scale contain many vertices with zero edges—for example, for 15,360 nodes at Scale 37, more than half the vertices have zero edges. Finally, our method (1) improves upon (4), indicating that vertex reordering has notable merit in improving the locality.

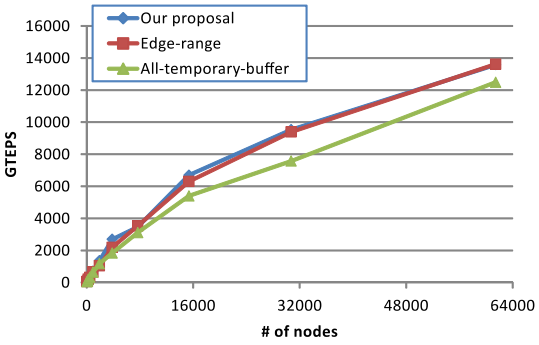

Fig. 18 Effects of hybrid load balancing on top-down BFS

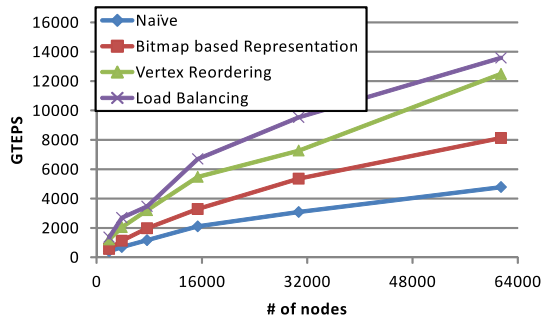

Fig. 19 Cumulative effect of all the proposed optimizations

Next, we investigate the effects of load balancing in top-down BFS. Figure 18 shows the results, where *"edge-range"* is using the algorithm in Fig. 13, whereas *"all-temporary-buffer"* is using the algorithm in Fig. 10, and *"Our proposal"* is the hybrid of the both. In the hybrid algorithm, the longer edge list is processed with the algorithm Fig. 13 and the shorter edge list is processed with the algorithm Fig. 10. We set the threshold for the length of the edge list to 1000. At some node sizes, the performance of *"Edge-range"* is almost identical to our proposed hybrid method. But this hybrid method performs best of those three methods.

Figure 19 shows the cumulative effect of all the optimization. The naive version uses DCSR without vertex reordering, and load-balanced using the algorithm in Fig. 10. By applying all the optimizations we have

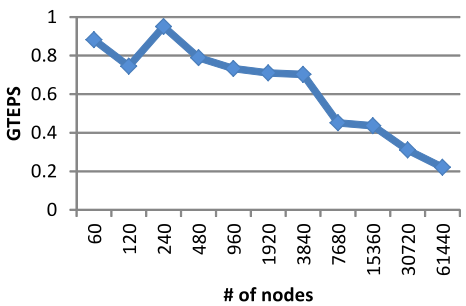

Fig. 20 Per-node execution performance in weak scaling

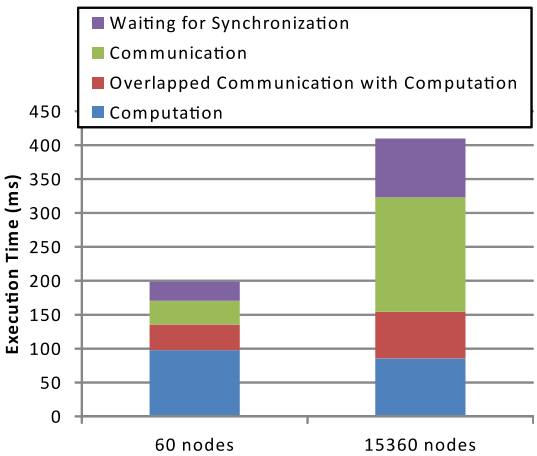

Fig. 21 Breakdown of performance numbers, 60 nodes versus 15,360

presented, we achieve 3.19 times speedup over the original version.

Figure 20 shows the per-node performance of weak scaling our proposed algorithm, where it slowly degrades as we scale the problem. Figure 21 shows the breakdown of time spent per each BFS for 60 and 15,360 nodes, exhibiting that the slowdown is largely due to increase in communication, despite various communication optimizations. This demonstrates that, even with an interconnect as fast as the K-Computer, network is still the bottleneck for large graphs, and as such, further hardware and algorithmic improvements are desirable for future extreme graph processing.

6.2 Using the Entire K-Computer

By using the entire K-Computer, we were able to obtain 38,621.4 GTEPS using 82,944 nodes and 663,552 cores with a Scale 40 problem in June 2015. This bested the previous record of 23,751 GTEPS recorded by LLNL's Sequoia BlueGene/Q supercomputer, with 98,304 nodes and 1,572,864 cores with a Scale 41 problem.

In the Tofu network of the K-Computer, a position in a six-dimensional mesh/torus network is given by six-dimensional coordinates, x, y, z, a, b, c. The x- and y-axes are coordinate axes that connect racks, and the length of the

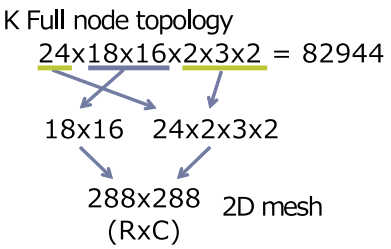

Fig. 22 Coordinates of K-Computer

x- and y-axes corresponds to the scale of the system. The z- and b-axes connect system boards, and the a- and c-axes are coordinate axes with a length of 2 that connect processors on each system board [1, 15]. For 2-D partitioning of a graph, $yz \times xabc$ is used instead of $xyz - abc$ and then we can obtain 288×288 for balanced 2-D mesh. The value for each coordinate is shown in Fig. 22 to fully leverage the full nodes of the K-Computer and balance the value of R and C in 2-D partitioning.

By all means, it is not clear whether we have hit the ultimate limit of the machine, i.e., whether or not we can tune the efficiency any further just by algorithmic changes. We know that BFS algorithm used for Sequoia is quite different from our proposed one, and it would be interesting to compare the algorithms vs. machines effect by cross-execution of the two (our algorithm on Sequoia and LLNL's algorithm on the K-Computer) and conducting a detailed analysis of both to investigate further optimization opportunities.

7 Related Work

As we mentioned, Yoo [16] proposed an effective method for 2-D graph partitioning for BFS in a large-scale distributed-memory computing environment; the base algorithm itself was a simple top-down BFS and was evaluated on a large-scale environment 32,768 node BlueGene/L.

Buluc et al. [5] conducted extensive performance studies of partitioning schemes for BFS on large-scale machines at LNBL, Hopper (6,392 nodes) and Franklin (9,660 nodes), comparing 1-D and 2-D partitioning strategies. Satish et al. [10] proposed an efficient BFS algorithm on commodity supercomputing clusters consisting of Intel CPU and the Infiniband Network. Checconi et al. [7] proposed an efficient parallel-distributed BFS on BlueGene using a communication method called *"wave"* that proceeds independently along the rows of the virtual processor grids. All the efforts here, however, use a top-down approach only as the underlying algorithm and are fundamentally at a disadvantage for graphs such as the Graph500 Kronecker graph whose diameter is relatively small compared to its size, as many real-world graphs are.

Hybrid BFS by Beamer [2] is the seminal work that solves this problem, on which our work is based. Efficient parallelization in a distributed-memory environment on a supercomputer is much more difficult and includes the early work by Beamer [3] and the work by Checconi [6] which uses a 1-D partitioning approach. The latter is very different to ours, not only in the difference in partitioning being 1-D compared to our 2-D, but also in taking advantage of the simplicity in ingeniously replicating the vertices with large number of edges among all the nodes, achieving very good overall load balancing. Performance evaluation on BlueGene/Q 65536 nodes has achieved 16,599 GTEPS, and it would be interesting to consider utilizing some of the strategies in our work.

8 Conclusion

For many graphs we see in the real world, with relatively small diameter compared to its size, hybrid BFS is known to be very efficient. The problem has been that, although various algorithms have been proposed to parallelize the algorithm in a distributed-memory environment, such as the work by Beamer [3] using 2-D partitioning, the algorithms failed to scale or be efficient for modern machines with tens of thousands of nodes and million-scale cores, due to the increase in memory and communication requirements overwhelming even the best machines. Our proposed hybrid BFS algorithm overcomes such problems by combination of various new techniques, such as bitmap-based sparse matrix representation, reordering of vertex ID, as well as new methods for communication optimization and load balancing. Detailed performance on the K-Computer revealed the effectiveness of each of our approach, with the combined effect of all achieving over $3\times$ speedup over previous approaches, and scaling to the entire 82,944 nodes of the machine effectively. The resulting performance of 38,621.4 GTEPS allowed the K-Computer to be ranked No. 1 on the Graph500 in June 2015 by a significant margin, and it has retained this rank to this date as of June 2016. We hope to further advance the optimizations to other graph algorithms, such as SSSP, on large-scale machines.

Acknowledgements This research was supported by the Japan Science and Technology Agency's CREST project titled "Development of System Software Technologies for post-Peta Scale High Performance Computing."

References

1. Ajima Y, Takagi Y, Inoue T, Hiramoto S, Shimizu T (2011) The tofu interconnect. In: 2011 IEEE 19th Annual Symposium on High Performance Interconnects, pp 87–94. doi:10.1109/HOTI.2011.21
2. Beamer S, Asanović K, Patterson D (2012) Direction-optimizing breadth-first search. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and

Analysis, SC '12, pp 12:1–12:10. IEEE Computer Society Press, Los Alamitos, CA, USA. http://dl.acm.org/citation.cfm?id=2388996.2389013
3. Beamer S, Buluc A, Asanovic K, Patterson D (2013) Distributed-memory breadth-first search revisited: Enabling bottom-up search. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW '13, pp 1618–1627. IEEE Computer Society, Washington, DC, USA. doi:10.1109/IPDPSW.2013.159
4. Buluc A, Gilbert JR (2008) On the representation and multiplication of hypersparse matrices. In: IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, pp 1–11. doi:10.1109/IPDPS.2008.4536313
5. Buluç A, Madduri K (2011) Parallel breadth-first search on distributed-memory systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pp 65:1–65:12. ACM, New York, NY, USA. doi:10.1145/2063384.2063471
6. Checconi F, Petrini F (2014) Traversing trillions of edges in real time: graph exploration on large-scale parallel machines. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp 425–434. doi:10.1109/IPDPS.2014.52
7. Checconi F, Petrini F, Willcock J, Lumsdaine A, Choudhury AR, Sabharwal Y (2012) Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pp 13:1–13:12. IEEE Computer Society Press, Los Alamitos, CA, USA. http://dl.acm.org/citation.cfm?id=2388996.2389014
8. Graph500: http://www.graph500.org/
9. Montagne E, Ekambaram A (2004) An optimal storage format for sparse matrices. Inf Process Lett 90(2):87–92. doi:10.1016/j.ipl.2004.01.014
10. Satish N, Kim C, Chhugani J, Dubey P (2012) Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp 1–11. doi:10.1109/SC.2012.70
11. Suzumura T, Ueno K, Sato H, Fujisawa K, Matsuoka S (2011) Performance characteristics of graph500 on large-scale distributed environment. In: Proceedings of the 2011 IEEE International Symposium on Workload Characterization, IISWC '11, pp 149–158. IEEE Computer Society, Washington, DC, USA. doi:10.1109/IISWC.2011.6114175
12. Ueno K, Suzumura T (2012) Highly scalable graph search for the graph500 benchmark. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC '12, pp 149–160. ACM, New York, NY, USA. doi:10.1145/2287076.2287104
13. Ueno K, Suzumura T (2013) Parallel distributed breadth first search on GPU. In: 20th Annual International Conference on High Performance Computing, pp 314–323. doi:10.1109/HiPC.2013.6799136
14. Yasui Y, Fujisawa K (2014) Fast and energy-efficient Breadth-First Search on a single NUMA system. Springer International Publishing, Cham. doi:10.1007/978-3-319-07518-1_23
15. Yokokawa M, Shoji F, Uno A, Kurokawa M, Watanabe T (2011) The k computer: Japanese next-generation supercomputer development project. In: Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design, ISLPED '11, pp 371–372. IEEE Press, Piscataway, NJ, USA. http://dl.acm.org/citation.cfm?id=2016802.2016889
16. Yoo A, Chow E, Henderson K, McLendon W, Hendrickson B, Catalyurek U (2005) A scalable distributed parallel breadth-first search algorithm on bluegene/l. In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05, pp 25. IEEE Computer Society, Washington, DC, USA. doi:10.1109/SC.2005.4