Performance of the Supercomputer Fugaku for Breadth-First Search in Graph500 Benchmark

Nakao, Masahiro
RIKEN Center for Computational Science

Ueno, Koji
Fixstars Corporation

Fujisawa, Katsuki
Institute of Mathematics for Industry, Kyushu University

Kodama, Yuetsu
RIKEN Center for Computational Science

他

https://hdl.handle.net/2324/4771851

出版情報：Lecture Notes in Computer Science: International Conference on High Performance Computing, ISC High Performance 2021: High Performance Computing. 12728, pp.372-390, 2021-06-17. Springer
バージョン：
権利関係：

Performance of the Supercomputer Fugaku for Breadth-First Search in Graph500 Benchmark

Masahiro Nakao[1][0000−0001−7848−1172], Koji Ueno[2], Katsuki Fujisawa[3],
Yuetsu Kodama[1], and Mitsuhisa Sato[1]

[1] RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo 650-0047, Japan
{masahiro.nakao, yuetsu.kodama, msato}@riken.jp https://www.r-ccs.riken.jp
[2] Fixstars Corporation, 1-11-1 Osaki, Shinagawa-ku, Tokyo 141-0032, Japan
kojiueno5@gmail.com https://www.fixstars.com
[3] Institute of Mathematics for Industry, Kyushu University, 744 Motooka, Nishi-ku,
Fukuoka 819-0395, Japan
fujisawa@imi.kyushu-u.ac.jp https://www.imi.kyushu-u.ac.jp

Abstract. In this paper, we present the performance of the supercomputer Fugaku for breadth-first search (BFS) problem in the Graph500 benchmark, which is known as a ranking benchmark used to evaluate large-scale graph processing performance on supercomputer systems. Fugaku is a huge-scale Japanese exascale supercomputer that consists of 158,976 nodes connected by the Tofu interconnect D (TofuD). We have developed a BFS implementation that can extract the performance of Fugaku. We also optimize the number of processes per node, one-to-one communication, performance power ratio, and process mapping in the six-dimensional mesh/torus topology of TofuD. We evaluate the BFS performance for a large-scale graph consisting of about 2.2 trillion vertices and 35.2 trillion edges using the whole Fugaku system, and achieve 102,956 giga-traversed edges per second (GTEPS), resulting in the first position of Graph500 BFS ranking in November 2020. This performance is 3.3 times higher than that of Fugaku's previous system, the K computer.

Keywords: Breadth-first search · Performance evaluation · Graph500.

1 Introduction

There is an increasing demand for computer systems capable of converting large-scale real-world data into a graph, which is a data structure representing relationships between elements with vertices and edges, and processing it at high speed. The graph processing is used in various fields for the analysis of connections between social network users, the optimization of very large scale integration (VLSI) layouts and road networks, whole-brain simulation, Internet of Things (IoT), search engines, drug discovery, gene analysis, and so on[7, 11, 16, 17]. In such cases, the number of vertices can exceed 1 trillion, and the number of edges can be several tens of times the number of vertices.

Table 1: Specifications of the supercomputer Fugaku and the K computer

Name	Supercomputer Fugaku	The K computer
CPU	A64FX, 48+2/4cores, 2.0/2.2GHz, 3,072/3,379GFlops (double precision)	SPARC64 VIIIfx, 8cores, 2.0GHz, 128GFlops (double precision)
Memory	HBM2, 32GB, 1,024GB/s	DDR3 SDRAM, 16GB, 64GB/s
Network	TofuD, 0.49 to 0.54μs (Latency) 6.8GB/s (Bandwidth)	Tofu, 0.91 to 1.15μs (Latency) 5.0GB/s (Bandwidth)
Nodes	158,976	82,944

Against this background, Graph500, a project for evaluating large-scale graph processing performance, has been ongoing since 2010 and released new listings of the top-performing systems twice-yearly (June and November)[1, 12]. In Graph500, a scale-free graph called Kronecker graph[8] is used. The term scale-free describes a property in which some vertices are connected to many other vertices while numerous others are connected to only a few vertices. Social network data are known to have a similar property. The Graph500 benchmark consists of breadth-first search (BFS) and single-source shortest path (SSSP). This paper focuses on BFS, which is a crucial algorithm used in the strongly connected component decomposition and centrality analysis of graphs.

The K computer [6] was ranked first in Graph500 for nine consecutive terms until June 2019, and it was removed from Graph500 following the decommissioning of the K computer. And then, the supercomputer Fugaku (Fugaku)[10], which is the successor of the K computer, has been ranked first since June 2020. This paper describes the BFS algorithm used for the Graph500 submission and the experimental evaluation results conducted on Fugaku.

The remainder of this paper is structured as follows. Section 2 provides an overview of Fugaku. Section 3 describes the Hybrid-BFS algorithm commonly used in Graph500. Section 4 introduces the BFS algorithm based on the Hybrid-BFS. Section 5 describes how we tune the performance of BFS. Section 6 discusses the evaluation of BFS on Fugaku. Section 7 summarizes this paper and discusses our future work.

2 The supercomputer Fugaku

Fugaku is a supercomputer installed at the RIKEN Center for Computational Science in Japan, and is scheduled to commence operation in 2021. Table 1 shows the specification of Fugaku. Each node has a single Fujitsu A64FX processor (A64FX)[10]. Fig. 1 shows the block diagram of A64FX. Fugaku consists of “compute node” and “compute node with IO node”. While the “compute node” performs calculations, the “compute node with IO node” performs both calculations and input/output processings. A64FX has 48 compute cores, while the “compute node” and “compute node with IO node” use two and four assistant cores, respectively. The assistant core deals with interruptions caused by

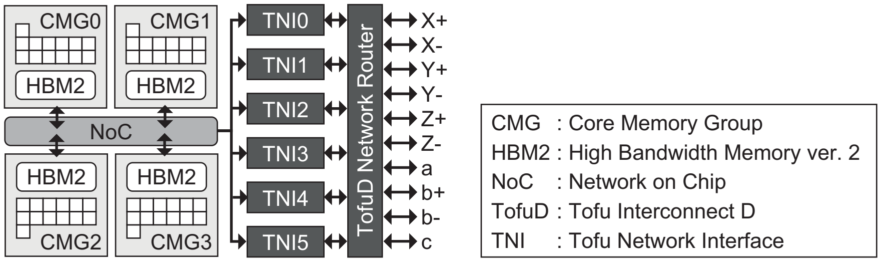


Fig. 1: A64FX processor[20]

OS, communications, and so on. The clock frequency of the A64FX core can be set to either 2.0 GHz or 2.2 GHz for each job depending on the user's preferences. The peak performance of double precision is 3,072 GFlops at 2.0 GHz and 3,379 GFlops at 2.2 GHz. A64FX consists of four Core Memory Groups (CMGs), each of which has 12 compute cores, a single assistant core, and an 8 GB High Bandwidth Memory ver. 2 (HBM2). The four CMGs are connected via a Network on Chip (NoC). The Fugaku interconnect uses Tofu Interconnect D (TofuD)[20]. The topology of TofuD is a six-dimensional mesh/torus in which the node position is specified by $XYZabc$ axes. Since the size of Fugaku is $(X, Y, Z, a, b, c) =$ (24, 23, 24, 2, 3, 2), the total number of nodes is 158,976. Also, A64FX has ten ports for TofuD, each $XYZb$ axis uses two ports, and each ac axis uses one port because ac axes consist of two nodes. The latency (8 bytes put communication) of Fugaku is 0.49 to 0.54 μs[20]. A64FX has six Tofu Network Interfaces (TNIs) and can communicate at 6.8 GB/s in six directions simultaneously. Thus, the injection bandwidth of each node is 40.8 GB/s.

Table 1 also shows the specification of the K computer for comparison. The peak performance of A64FX at 2.2 GHz is 26.4 times, the memory capacity is twice, and the memory bandwidth is 16.0 times that of the K computer. The network interconnect used in the K computer is Tofu Interconnect (Tofu)[19], which is the predecessor of TofuD. While its topology is the same as TofuD, the size of the K computer is $(X, Y, Z, a, b, c) =$ (24, 18, 16, 2, 3, 2). Since the total number of nodes is 82,944, the number of nodes in Fugaku is 1.9 times that of the K computer. The latency of Fugaku is about half and the network bandwidth of Fugaku is 1.4 times that of the K computer. Since the K computer had four TNIs in each node, the injection bandwidth is 20.0 GB/s. Thus, the injection bandwidth of Fugaku is 2.0 times that of the K computer.

3 Hybrid-BFS for large-scale system

3.1 Algorithm for shared memory system

Fig. 2 shows an overview of Hybrid-BFS[13] where BFS is executed while switching between the conventional search method called "top-down approach" and another search method called "bottom-up approach". The current starting

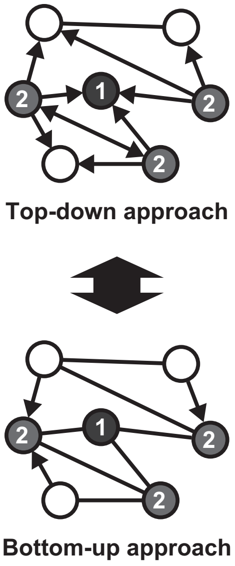


Fig. 2: Overview of Hybrid-BFS

Algorithm 1: Hybrid-BFS

```
1  hybrid-bfs(vertices, source, nbr)
2    frontier ← {source}
3    next ← {}
4    parents ← [-1,-1,...,-1]
5    while frontier ≠ {} do
6    | if next-direction(...) = top-down then
7    | | top-down(vertices, frontier, next, parents, nbr)
8    | else
9    | | bottom-up(vertices, frontier, next, parents, nbr)
10   | frontier ← next
11   | next ← {}
12   return parents
13
14 top-down(vertices, frontier, next, parents, nbr)
15   for v ∈ frontier do
16   | for n ∈ nbr[v] do
17   | | if parents[n] = -1 then
18   | | | parents[n] ← v
19   | | | next ← next ∪ {n}
20
21 bottom-up(vertices, frontier, next, parents, nbr)
22   for v ∈ vertices do
23   | if parents[v] = -1 then
24   | | for n ∈ nbr[v] do
25   | | | if n ∈ frontier then
26   | | | | parents[v] ← n
27   | | | | next ← next ∪ {v}
28   | | | | break
```

points are ②, looking for unsearched adjacencies. The issue with the top-down approach is that current start points must check all adjacencies. Since most adjacencies have been searched (the first start point ① and current start points ② have been searched), redundant checks occur frequently. Therefore, in the bottom-up approach, the search is performed in the opposite direction to the top-down approach, in which the current start points ② are searched from the unsearched vertices (◯ in the figure). The advantage of the bottom-up approach is that if even one of the current start points ② is found, the check can be terminated, reducing redundant checks.

Algorithm 1 shows the pseudo-code of the Hybrid-BFS. In line 2, the first starting point (*source*) is substituted for the visited points set (*frontier*). In line 3, the next visitation point set (*next*) is initialized as an empty set. In line 4, BFS tree (*parents*) for the final output, is initialized. Note that the substitution of "-1" for *parents* means that a vertex has not yet been visited.

The top-down approach in the function **top-down()** first checks whether the vertices adjacent to *frontier* have been visited (lines 15–17). Note that *nbr (neighbors)* is an adjacent set of vertices. If unvisited, the connection source of an unvisited vertex is assigned to *parents* (line 18). Additionally, the unvisited vertices are added to *next* without duplication (line 19). In the top-down

$$\begin{matrix} A_{1,1} & A_{1,2} & \cdots & A_{1,C} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,C} \\ \vdots & \vdots & \ddots & \vdots \\ A_{R,1} & A_{R,2} & \cdots & A_{R,C} \end{matrix}$$

Fig. 3: Distribution of adjacency matrix

Algorithm 2: Parallel top-down approach

```
parallel-top-down(...)
  $f \leftarrow$ {source}
  $n \leftarrow \{\}$
  $\pi \leftarrow$ [-1,-1,...,-1]
  for all processes $P(i,j)$ in parallel do
    while $f \neq \{\}$ do
      transpose-vector($f_{i,j}$)
      $f_i \leftarrow$ allgatherv($f_{i,j}$, $P(:,j)$)
      $t_{i,j} \leftarrow \{\}$
      for $u \in f_i$ do
        for $v \in A_{i,j}(:,u)$ do
          $t_{i,j} \leftarrow t_{i,j} \cup (u,v)$
      $t_{i,j} \leftarrow$ alltoallv($t_{i,j}, P(i,:)$)
      for $(u,v) \in t_{i,j}$ do
        if $\pi_{i,j}(v) = -1$ then
          $\pi_{i,j}(v) \leftarrow u$
          $n_{i,j} \leftarrow n_{i.j} \cup v$
      $f \leftarrow n$
      $n \leftarrow \{\}$
  return $\pi$
```

approach, vertices in *frontier* are used as the starting points in searches for unvisited vertices adjacent to them. In contrast, in the bottom-up approach of the function **bottom-up()**, all unvisited vertices are used as the starting points and the searches determine whether the vertices adjacent to them belong to *frontier* (lines 22–25). When a vertex belonging to *frontier* is found, it is assigned to *parents* and its starting point is added to *next* without duplication (lines 26–27).

The advantage of the bottom-up approach is that when one vertex belonging to *frontier* is found, the search for that starting vertex can be terminated (line 28), thus reducing the redundant checks seen in the top-down approach. However, since the bottom-up approach requires checking whether all vertices have been visited, the top-down approach is faster when *frontier* is small. Therefore, the Hybrid-BFS uses the top-down approach when *frontier* is small, and the bottom-up approach when *frontier* is large. Although we have omitted the full details here, the **next-direction()** function in line 6 dynamically decides whether to switch between the top-down and bottom-up approaches.

3.2 Algorithm for distributed memory system

To handle large graphs, the parallel Hybrid-BFS has been proposed[14]. In the parallel Hybrid-BFS, the adjacency matrix A is assigned to the processes divided into two dimensions (R rows and C columns) as shown in Fig. 3. A process $P(i,j)$ has information on a partial adjacency matrix $A_{i,j}$. Algorithms 2 and 3 show the pseudo-codes for the parallel top-down and bottom-up approaches, respectively. The parallel Hybrid-BFS is executed by switching the approaches, as well as the Hybrid-BFS in Algorithm 1. The f, n, and π correspond to $frontier$, *next*, and *parents*, respectively. The t is a sparse vector for temporarily holding two

Algorithm 3: Parallel bottom-up approach

```
1  parallel-bottom-up(...)
2    $f \leftarrow$ {source}
3    $c \leftarrow$ {source}
4    $n \leftarrow \{\}$
5    $\pi \leftarrow$ [-1,-1,...,-1]
6    for all processes $P(i,j)$ in parallel do
7    | while $f \neq \{\}$ do
8    | | transpose-vector($f_{i,j}$)
9    | | $f_i \leftarrow$ allgatherv($f_{i,j}$, $P(:,j)$)
10   | | for $s$ in 0 .. $C$-1 do
11   | | | $t_{i,j} \leftarrow \{\}$
12   | | | for $u \in c_{i,j}$ do
13   | | | | for $v \in A_{i,j}(u,:)$ do
14   | | | | | if $v \in f_i$ then
15   | | | | | | $t_{i,j} \leftarrow t_{i,j} \cup (u,v)$
16   | | | | | | $c_{i,j} \leftarrow 1$
17   | | | | | | break
18   | | | $t_{i,j} \leftarrow$ sendrecv($t_{i,j}, P(i,j+s), P(i,j$-$s)$)
19   | | | for $(v,u) \in t_{i,j}$ do
20   | | | | $\pi_{i,j}(v) \leftarrow u$
21   | | | | $n_{i,j} \leftarrow n_{i.j} \cup v$
22   | | | $c_{i,j} \leftarrow$ sendrecv($c_{i,j}, P(i,j+1), P(i,j$-1$)$)
23   | | $f \leftarrow n$
24   | | $n \leftarrow \{\}$
25   return $\pi$
```

row-starts	0 2 2 2 2 2 2 3 4
dst	4 5 3 1

Compressed Sparse Row (CSR)

row-starts	0 2 3 4
bitmap	1 0 0 0 0 0 1 1
offset	0 1 3
dst	4 5 3 1

Bitmap-based CSR (BCSR)

Fig. 4: Compressed formats

adjacent vertices (u and v). The c is a bitmap of checked vertices, while the f and n are sparse vectors for the top-down approach and bitmaps for the bottom-up approach, respectively. The π is a dense vector in both approaches.

In Algorithm 2, in lines 7–8, the f is shared in the column process. In lines 9–13, information about the f and the adjacent vertices is exchanged in the row process. In lines 14–17, the π and n are created. In Algorithm 3, lines 8–9 are the same as lines 7–8 of Algorithm 2, except for the data structure of the f. The **for** statement in lines 10–22 is divided into C sub-steps. The reason is to reduce the number of vertices to be searched for in each process by periodically updating c in the row process in line 22, thereby improving the overall speed. In lines 10–18, the information on unvisited vertices adjacent to the f is exchanged in the row process. In lines 19–21, the π and n are created.

4 Improvement to Hybrid-BFS

This section introduces the BFS algorithm for Fugaku, which is also adopted in the K computer[9]. Since the algorithm is an improved version of the Hybrid-BFS described in Section 3, this section describes only the changes.

Table 2: Memory consumption in CSR and BCSR

	CSR		BCSR	
	Order	Actual	Order	Actual
row-starts	$n'C$	2048MB	$n'p$	190MB
bitmap	-	-	$n'C/64$	32MB
offset	-	-	$n'C/64$	32MB
dst	$n'd$	1020MB	$n'd$	1020MB
TOTAL	$n'(C+d)$	3068MB	$n'(\frac{C}{32}+p+d)$	1274MB

4.1 Bitmap-based representation for adjacency matrix

When using a conventional compressed sparse row (CSR) as a format for storing an adjacency matrix, the array *dst*, which holds the output vertex number, and the offset array *row-starts* of the edge vertex numbers are used. For efficient edge information retrieval, the smaller *row-starts* size is desirable. However, the size of *row-starts* is proportional to C in the case of a two-dimensional division of R rows and C columns.

To resolve the issue, Bitmap-based CSR (BCSR) is proposed, which can extract edge information more efficiently and with less memory than CSR. BCSR provides the following features: (1) Compress the *row-starts* in CSR so that only the edge start position of a vertex with one or more edges is retained. (2) Use the *bitmap*, which is an array of bits per vertex that indicates whether each vertex has at least one edge. (3) Use the array *offset* to efficiently calculate the vertex number of an edge source. The position of *row-starts* at a vertex is the number of bits standing from the beginning of the *bitmap* to the bit corresponding to the vertex. To efficiently calculate the number of standing bits in *bitmap*, the cumulative total of bits is stored at *offset* in advance, word by word.

Fig. 4 shows examples of CSR and BCSR when the edge list is {(0, 4), (0, 5), (6, 3), (7, 1)} where each word is assumed to be 4 bits for the sake of explanation. The *row-starts* in CSR is represented in BCSR as three arrays: *row-starts*, *bitmap*, and *offset*. Next, Table 2 shows a comparison of the amount of memory where one word is set to 64 bits. Here, n' is the number of vertices per node, d is the degree, and p is the probability of having one or more edges in a row from a partial adjacency matrix of a process. Table 2 also shows the actual memory usage using a Kronecker graph used in Graph500 with 16 billion vertices and 256 billion edges when the two-dimensional division of $R \times C = 64 \times 32$. This result indicates that BCSR is more memory-efficient than CSR.

4.2 Sorting of vertex number

Bit positions in the bitmap are generally in vertex number order. A Kronecker graph has vertices with large and small degrees, and the vertices with larger degrees are accessed more frequently. Thus, the memory locality can be improved

$$\begin{array}{cccc}
A^{(1)}_{1,1} & A^{(1)}_{1,2} & \cdot\cdot & A^{(1)}_{1,C} \\
A^{(1)}_{2,1} & A^{(1)}_{2,2} & \cdot\cdot & A^{(1)}_{2,C} \\
\vdots & \vdots & \ddots & \vdots \\
A^{(1)}_{R,1} & A^{(1)}_{R,2} & \cdot\cdot & A^{(1)}_{R,C} \\
A^{(2)}_{1,1} & A^{(2)}_{1,2} & \cdot\cdot & A^{(2)}_{1,C} \\
\vdots & \vdots & \ddots & \vdots \\
A^{(C)}_{R,1} & A^{(C)}_{R,2} & \cdot\cdot & A^{(C)}_{R,C}
\end{array}$$

Fig. 5: Distribution of adjacency matrix by Yoo[4]

Algorithm 4: Simple thread parallelization

```
1 top-down-sender-naive($f_i$, $A_{i,j}$)
2   for $u \in f_i$ in parallel do
3   | for $v \in A_{i,j}(:, u)$ do
4   | | $k \leftarrow$ owner($v$)
5   | | $t_{i,j,k} \leftarrow t_{i,j,k} \cup (u, v)$
```

Algorithm 5: Proposed thread parallelization

```
1 top-down-sender-load-balanced($f_i$, $A_{i,j}$)
2   for $u \in f_i$ in parallel do
3   | for $k \in P(i, :)$ do
4   | | $(v_0, v_1) \leftarrow$ edge-range($A_{i,j}(:, u), k$)
5   | | $r_{i,j,k} \leftarrow r_{i,j,k} \cup (u, v_0, v_1)$
6   for $k \in P(i, :)$ in parallel do
7   | for $(u, v_0, v_1) \in r_{i,j,k}$ do
8   | | for $v \in A_{i,j}(v_0{:}v_1, u)$ do
9   | | | $t_{i,j,k} \leftarrow t_{i,j,k} \cup (u, v)$
```

by arranging the bit positions in degree order. In the algorithm, vertex numbers are reassigned in degree order within the process. Note that *parents* is created in degree order with the technique. Thus, it prepares a new array that holds the original vertex numbers and is used for writing to *parents*.

4.3 Yoo's distribution of adjacency matrix

When applying the distribution shown in Fig. 3 to the adjacency matrix, communication in **transpose-vector()** is required to transpose $frontier$ shown in line 7 of Algorithm 2 and line 8 of Algorithm 3. By applying the distribution proposed by Yoo[4], the communications can be removed. Fig. 5 shows the distribution. The distribution in the rows is the same as Fig. 3, while the distribution in the columns is $R \times C$ block-cyclic distribution.

4.4 Load balancing in top-down approach

Algorithm 4 shows a simple example of thread implementation in lines 10–12 of Algorithm 2. In line 2, it is threaded by the input source vertices in $frontier$. In line 3, $A_{i,j}(:, u)$ is an edge list whose edge input source is u. In line 4, the **owner(**v**)** function returns the process in charge of the output destination vertex v. In line 5, the adjacent vertex information is stored. Although the technique is simple, a large load imbalance between threads may occur because the degree of a Kronecker graph differs significantly depending on the vertex.

To resolve this load imbalance, it is threaded by the output destination vertices. Algorithm 5 shows the technique which uses two thread-parallelized **for** statements. The first **for** statement stores the information of the output destination vertices for each process in charge, and the second **for** statement stores the

Algorithm 6: Proposed bottom-up approach

```
**parallel-bottom-up**(...)
  $f \leftarrow$ {source}
  $c \leftarrow$ {source}
  $n \leftarrow \{\}$
  $\pi \leftarrow$ [-1,-1,...,-1]
  **for** all processes $P(i,j)$ **in parallel do**
    **while** $f \neq \{\}$ **do**
      $f_i \leftarrow$ **allgatherv**($f_{i,j}$, $P(:,j)$)
      **for** $s$ **in** 0 .. $C$-1 **do**
        $t_{i,j} \leftarrow \{\}$
        **for** $u \in c_{i,j}$ **do**
          **for** $v \in A_{i,j}(u,:)$ **do**
            **if** $v \in f_i$ **then**
              $t_{i,j} \leftarrow t_{i,j} \cup (u,v)$
              $c_{i,j} \leftarrow 1$
              **break**
        $c_{i,j} \leftarrow$ **sendrecv**($c_{i,j}, P(i,j$+1$), P(i,j$-1$)$)
      $t_{i,j} \leftarrow$ **alltoallv**($t_{i,j}, P(i,:)$)
      **for** $(v,u) \in t_{i,j}$ **do**
        $\pi_{i,j}(v) \leftarrow u$
        $n_{i,j} \leftarrow n_{i.j} \cup v$
      $f \leftarrow n$
      $n \leftarrow \{\}$
  **return** $\pi$
```

Table 3: Communication costs

Approach	Pattern	Times	Words
Top-down	allgatherv	$O(1)$	nR
	alltoallv	$O(1)$	$4m$
Bottom-up	allgatherv	$O(1)$	$s_b nR/64$
	sendrecv	$O(C)$	$s_b nC/64$
	alltoallv	$O(1)$	$2n$

set of adjacent vertices. The function **edge-range(**$A_{i,j}(:,u)$**,** k**)** in line 4 returns the range of the edge list for which the process in charge is k.

In Algorithm 4, an adjacent vertex is not stored in the communication buffer, but in a temporary buffer in line 5. The reason is that the data need to be contiguous for communication but the number of elements to be sent to each process cannot be known in advance. In contrast, in Algorithm 5, the number of vertices passed to each process in the first **for** statement can be counted. Therefore, in line 9, the adjacent vertices are used for communication without the temporary buffer. However, the disadvantage of the technique is that the amount of information in r is larger than that in t. When searching for vertices whose degree is relatively small compared to the number of destination processes, the amount of data written to r is larger than that to t, which is inefficient. Accordingly, the techniques in Algorithm 4 and Algorithm 5 are switched depending on the degree of each vertex and the number of destination processes.

4.5 Communication in bottom-up approach

Use of collective communication In the **sendrecv** communication in line 18 of Algorithm 3, point-to-point communication is performed within the row process group. As a result of preliminary experiments in a large-scale environment, it was found that the communication efficiency deteriorates when such unscheduled communications occur frequently. Therefore, by using **alltoallv** communication instead of the **sendrecv** communication, data are exchanged collectively,

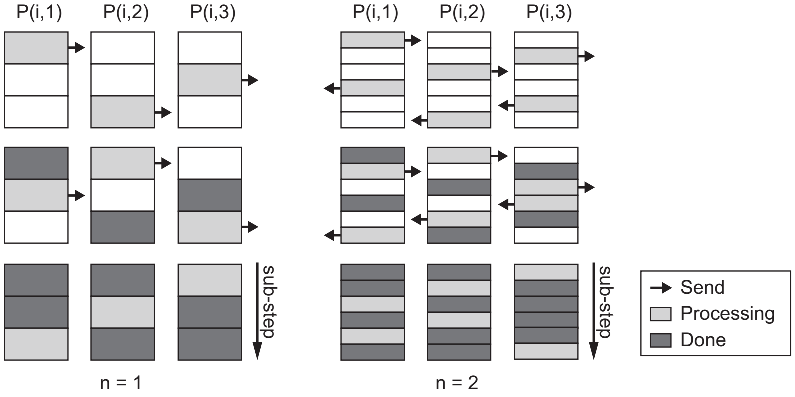


Fig. 6: Overlapping communication with computation

as shown in line 18 of Algorithm 6. Additionally, as described in Section 4.3, the **transpose-vector()** function is removed in Algorithm 6.

Switching data structure by vertex concentration Table 3 shows the communication costs of BFS. The third column shows the order of the communications required to perform one approach. The fourth column shows the communication size required to perform one BFS, assuming that the approach is not switched. Here, one word is 64 bits, n is the number of vertices, R and C are the sizes of each dimension of the process grid, m is the number of edges, and s_b is the number of times the bottom-up approach is performed. Additionally, for the sake of formula simplification, $(C-1)/C \approx 1$ is set and one-word communications are excluded. In the bottom-up approach, the one-step communication sizes of **allgatherv** and **sendrecv** increase in proportion to R and C, respectively. Note that **allgatherv** in the top-down approach uses a sparse vector and is executed only when $frontier$ is small, so there is no problem.

To reduce the communication size of the **allgatherv** and **sendrecv** in the bottom-up approach, a technique is used to select a bitmap or sparse vector according to the vertex concentration of the data automatically. When a sparse vector is used for each communication, the communication size of **allgatherv** is proportional to the number of vertices in $frontier$, and the communication size of **sendrecv** is proportional to the number of the unvisited vertices. In other words, when the number of vertices is smaller than $n/64$, the communication size of each can be reduced by using the sparse vector.

Overlapping communication with calculation To proceed with communication and calculation simultaneously in lines 9–17 of Algorithm 6, the sub-step in line 9 increases from C to $n \times C$. Our implementation uses $n = 4$. In addition, to effectively use torus topology networks such as TofuD, the **sendrecv** communication in line 17 is performed simultaneously in two directions. Fig. 6

shows its concept when $C = 3$. In the case of $n = 2$, communication to the right side, calculation process, and communication to the left side can be performed simultaneously. Note that $P(i, 1)$ and $P(i, 3)$ are directly connected in a torus topology. For reducing the communication waiting time, the processing order of the receiving process is the receiving order, not the loop order.

5 Performance optimization for Fugaku

This section reports how to optimize the BFS performance using up to 16,384 nodes, while the next Section 6 reports the final evaluation using more nodes. Note that these sections evaluate the BFS performance on Fugaku, but the evaluation results are not guaranteed to match the results at the start of sharing.

5.1 Graph500 benchmark

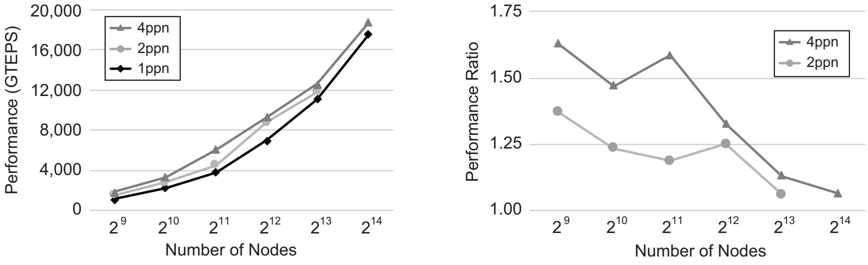
The number of vertices in a graph used in Graph500 is a power of two and is expressed as 2^{SCALE}. The number of edges is 16 times the number of vertices. The BFS performance unit is a traversed edges per second (TEPS)[1]. According to the Graph500 regulation[1], 64 vertices are randomly selected as the start points, after which BFS processing is performed on each. The harmonic mean of all 64 BFS performance values is set as the evaluation performance value. Since 64 times is excessive for the performance optimization performed in this section, the harmonic mean of 16 times in BFS is used as the performance value. In the next Section 6, the harmonic mean value produce by 64 BFS repetitions is used.

5.2 Setting parameters

In the evaluations, a graph size per node is set at $SCALE = 24$ and is measured with weak scaling. In Fugaku, users can specify one- to three-dimensional logical process layouts (job shapes). Since BFS uses the $R \times C$ two-dimensional process grid, we specify the two-dimensional job shape. In this case, each process is assigned to a node so that it has a physically two-dimensional torus topology. Please note that due to Fugaku's job scheduler, if the number of nodes used is 384 or less, it may not become the torus physically. Thus, in this experiment, 384 or more nodes will be used. Table 3 indicates that the communication size becomes smaller when the values of R and C are close. Note that if $R = C$ cannot be set, $R > C$ is desirable. Thus, if the number of processes is a square number, R and C should be set to the same value. If not a square number, R should be set to be larger and the difference between R and C should be set to be as small as possible. For example, if the number of processes is 8, then (R, C) = (4, 2).

5.3 Optimization of the number of processes per node

This section examines the optimum number of processes assigned to one node. The evaluation uses 1, 2, or 4 processes per node (denoted 1ppn, 2ppn, and 4ppn,



(a) Performance (b) Performance ratio with 1ppn

Fig. 7: Performance and performance ratio for each process with weak scaling

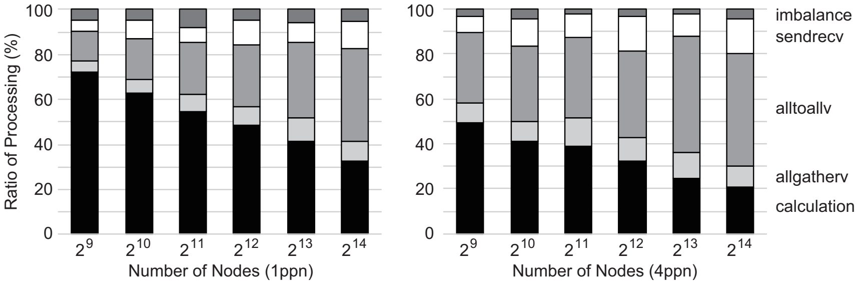
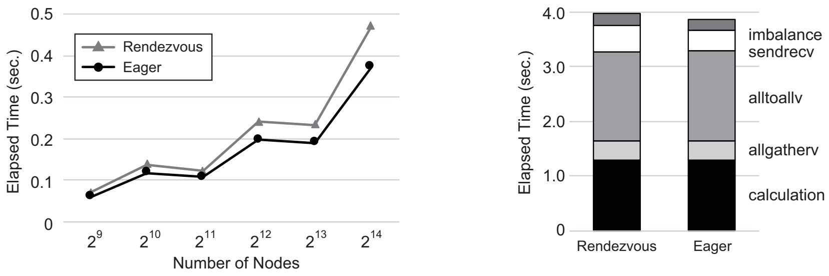


Fig. 8: Processing time ratio of Fig. 7

respectively) because A64FX has four CMGs shown in Fig. 1. The number of threads in each process is 48, 24, or 12.

Fig. 7a shows the performance results for each number of processes per node and Fig. 7b shows the relative performance of 2ppn and 4ppn when the result of 1ppn is 1.0. The result of 16,384 (=2^{14}) nodes for 2ppn could not be measured due to a system malfunction. The results of 16,384 nodes for 1ppn and 4ppn are 17,560 GTEPS and 18,738 GTEPS, respectively. Fig. 7 indicates that the performance is higher in the order of 4ppn, 2ppn, and 1ppn, but the performance difference becomes smaller as the number of nodes increases. One of the reasons for this performance difference is that at 1ppn and 2ppn, each thread frequently gets data across the CMGs in the process. In addition, 4ppn has a smaller data size per process, so the cache hit rate is higher. According to the profiler provided by Fugaku, the number of L2 misses in the case of 4ppn was about half that in the case of 1ppn.

Fig. 8 shows the time ratio of each BFS process for 1ppn and 4ppn. The **calculation** is the local processing, while **allgatherv**, **alltoallv**, and **sendrecv** are the communication times listed in Table 3. Additionally, **imbalance** is the synchronization waiting time when barrier synchronization is performed at the end of the approach. Fig. 8 indicates that the communication time ratio for 1ppn



(a) Elapsed time of **sendrecv**

(b) Breakdown in 2^{14} nodes

Fig. 9: Comparison of Rendezvous and Eager

is smaller than that for 4ppn. The reason is considered to be that the number of communication partners of 1ppn is less than that of 4ppn.

Although we used up to 16,384 nodes in this section, we will report evaluations using more nodes in Section 6. From an examination of Fig. 8, it can be predicted that the communication time ratio will be larger when a larger number of nodes is used. Additionally, in general, as the number of processes increases, the amount of memory consumed internally by the MPI library increases. Thus, the subsequent evaluations will be performed for 1ppn.

5.4 Use of Eager method

In the point-to-point communication of most MPI implementations, the Eager and Rendezvous methods are implemented. The Eager method sends a message via a buffer regardless of the state of the receiving process. In contrast, the Rendezvous method does not send a message until the receiving process is ready. Since the Eager method is suitable for small message communication, most MPI implementations switch the Eager and Rendezvous methods automatically depending on message size.

As shown in Table 3 and Fig. 6, point-to-point communication is performed in **sendrecv**. In the previous evaluation described in Section 5.3, we found that the Rendezvous method was used for all **sendrecv** communications. Here, it should be noted that the Fujitsu MPI library provided by Fugaku can change the switching threshold between the Eager and Rendezvous methods by setting a parameter in the "mpiexec" command. If the node on Fugaku has sufficient memory, the Eager method usage rate can be increased using the parameter. In this experiment, the threshold is set to 512,000 bytes.

This section evaluates the performance when the Eager method is used for all **sendrecv** communications. Fig. 9 shows the results. For comparison purposes, Fig. 9 also shows the results for 1ppn in Section 5.3 as the "Rendezvous" item. Fig. 9a shows the communication time of **sendrecv**, and Fig. 9b shows the breakdown when using 16,384 nodes. These results show that BFS performance is improved by using the Eager method. The result of 16,384 nodes using the

Eager method is 17,964 GTEPS. In Fig. 9a, the reason for the staircase shape of the measured value is its relationship to the value of C, shown in Table 3. For example, the values of (R, C) when using 2^{12}, 2^{13}, and 2^{14} nodes are (64, 64), (128, 64), and (128, 128), respectively.

In the subsequent evaluations, the switching threshold will be adjusted so that all **sendrecv** communications will use the Eager method.

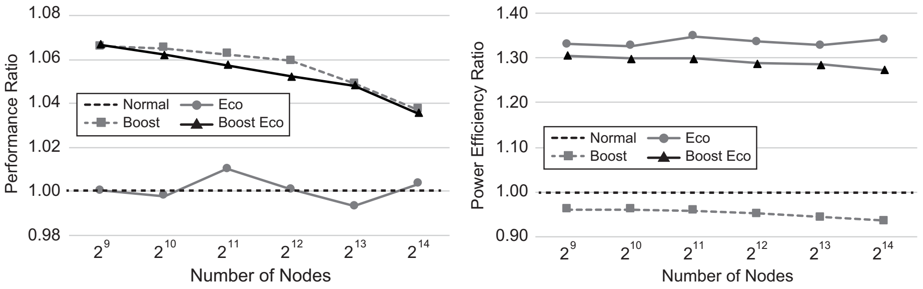
5.5 Power management

As mentioned in Section 2, the clock frequency of the A64FX core can be specified as either 2.0 or 2.2 GHz for each job. While the operation at 2.0 GHz is called "Normal mode", that at 2.2 GHz is called "Boost mode". Of course, Boost mode requires more power than Normal mode. To reduce power consumption, "Eco mode" is also available on A64FX. In Eco mode, the two floating-point arithmetic pipelines of A64FX are limited to one, and power control is performed according to the maximum power used at that time. Since BFS does not perform floating-point arithmetic, Eco mode can be expected to reduce power consumption without affecting performance. With that point in mind, this section reports on the performance and power consumption of BFS when using Boost mode and Eco mode. Since the modes are orthogonal settings, the evaluation is performed using the following four combinations:

- **Normal mode:** 2.0 GHz and two floating-point arithmetic pipelines (this mode was used in Sections 5.3 and 5.4).
- **Boost mode:** 2.2 GHz and two floating-point arithmetic pipelines
- **Eco mode:** 2.0 GHz and one floating-point arithmetic pipeline
- **Boost Eco mode:** 2.2 GHz and one floating-point arithmetic pipeline

There are two power measurement methods used in Fugaku. One is performed by a user (called user method), the other is performed by the facility (called facility method). The user method measures the power in a part of the user program using dedicated APIs on a node-by-node basis, whereas the facility method measures the entire job in rack units (384 nodes are stored in one rack), which means that nodes executing BFS must occupy the rack. In this section, power is measured using the user method. The difference is that the user method measures the direct current (DC) supplied from the power supply unit (PSU), while the facility method measures the 200 V alternating current (AC) supplied to the PSU. In a preliminary evaluation of three racks (1,152 nodes) using Normal mode, the power measured by the user method was found to be 117 kW, while the facility method measurement was 126 kW. The difference between these values is considered to be the AC/DC conversion loss plus the power of the control device in the rack that is not included in the node power[18].

Fig. 10a shows the performance ratio of the other modes to that of Normal mode, and Fig. 10b shows the corresponding power efficiency (TEPS/W) ratios. Thus, a value higher than 1.00 indicates performance or power efficiency better than that of Normal mode. Fig. 10a indicates that the performance is improved



(a) Performance comparison (b) Power efficiency comparison

Fig. 10: Comparison between modes

by about 4 to 7 % by setting Boost mode or Boost Eco mode, whereas the performance does not change when Eco mode is set. Fig. 10b indicates that the power efficiency is improved by 27 to 35 % by setting Eco mode or Boost Eco mode. From the above results, it can be said that Boost Eco mode is most suitable for BFS because it has both high performance and good power efficiency. In Boost Eco mode, the result for 16,384 nodes is 18,607 GTEPS in performance, 1,408 kW in power consumption, and 13.22 MTEPS/W in power efficiency.

5.6 Six-dimensional process mapping

As described in Section 5.2, it is desirable that R and C be close to each other. However, since the maximum size of two-dimensional job shapes supported by the Fugaku job scheduler is $YZc \times Xab$, it is 1,104 × 144 for the whole system, and the difference between R and C is 7.67 times. Therefore, we perform a process mapping that can set any combination of axes of the TofuD six-dimensional network to R and C. For example, in the case of the whole system, by assigning R to the XY axes and C to the $Zabc$ axes, 552 × 288 process grid is created. The difference between R and C is 1.92 times.

In the process mapping for C, since the **sendrecv** communication shown in Fig. 6 is suitable for adjacent communication, the mapping should ensure that all the nodes are adjacent. If not, performance will be degraded due to communication collisions. Fig. 11 shows an example of assigning the abc axes ($2 \times 3 \times 2$) to C. First, the assigned axis is expanded in two dimensions. The horizontal is the first axis, and the vertical is the remaining axes. Then, all processes are assigned so that they are adjacent to each other. To make the first and last processes (0 and 11) adjacent to each other physically, the topology of the last axis must be either a torus, or the a or c axis because the a and c axes consist of two nodes. Regarding the process mapping for R, it is not necessary to take the above measure because there is no adjacent communication.

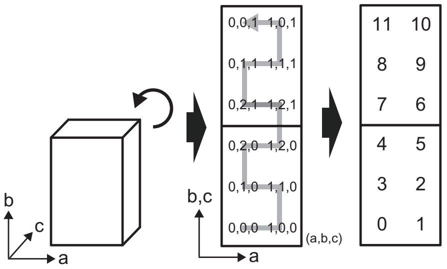


Fig. 11: Process mapping for C

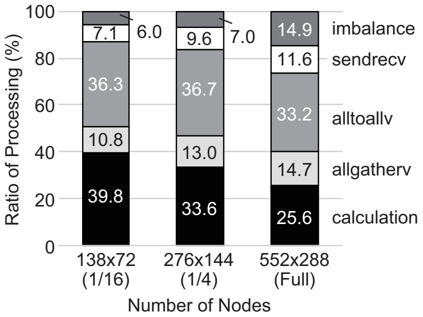


Fig. 12: Time ratio of processing

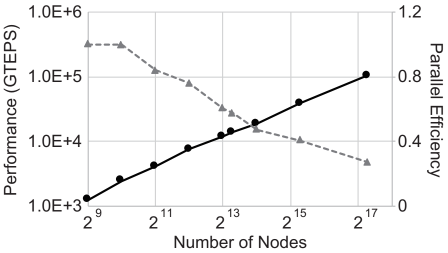


Fig. 13: Performance

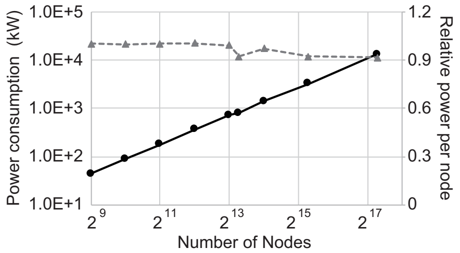


Fig. 14: Power consumption

6 Performance evaluation on Fugaku

6.1 Performance on whole Fugaku system

This section evaluates the BFS algorithm on the whole Fugaku system. As described above, we set (R, C) to (552, 288) = 158,976 nodes, $SCALE$ = 41 (a graph with 2^{41} vertices and 2^{45} edges), and Boost Eco mode. For a comparison purpose, we also conducted evaluations using 1/4 and 1/16 of Fugaku. We set (R, C) to (276, 144) = 39,744 nodes and $SCALE$ = 39 for 1/4 system, and (R, C) to (138, 72) = 9,936 nodes and $SCALE$ = 37 for 1/16 system. Fig. 12 shows the time ratio of each process in this evaluation. As the number of nodes increases, the ratio of total communication (**sendrecv** + **alltoallv** + **allgatherv**) and imbalance increase. The performance of each is 102,956 GTEPS for the whole system, 38,749 GTEPS for 1/4 system, and 13,738 GTEPS for 1/16 system. In addition, power consumption and power efficiency in the whole system measured by the facility method are 14,961 kW and 6.88 MTEPS/W, respectively.

Fig. 13 and Fig. 14 summarize the performance and power consumption results so far; they also show the parallel efficiency and relative power per node with the 2^9-node result set to 1. Note that for power consumption, all results are measured with the user method. As the number of nodes increases, Fig. 13 shows

Table 4: Graph500 list for June 2019 and November 2020

	June 2019			November 2020		
	NAME	SCALE	GTEPS	NAME	SCALE	GTEPS
1st	**K computer**	40	**31,302**	**Supercomputer Fugaku**	41	**102,956**
2nd	Sunway TaihuLight	40	23,756	Sunway TaihuLight	40	23,756
3rd	Sequoia	41	23,751	TOKI-SORA	36	10,813
4th	Mira	40	14,982	Summit	40	7,666
5th	SuperMUC-NG	39	6,279	SuperMUC-NG	39	6,279

a sharp drop in parallel efficiency, whereas Fig. 14 shows a slight decrease in relative power per node. The reason is considered to be that the communication load becomes large.

6.2 Comparison with other systems

Table 4 shows the first to fifth places of Graph500 in June 2019 and November 2020. In June 2019, the first place was the K computer; this was the last ranking prior to its decommissioning. In November 2020, Fugaku was ranked first based on the performance optimization described in this paper. The Fugaku performance value was 3.3 times that of the K computer and 4.3 times that of Sunway TaihuLight. Between June 2019 and November 2020, Sequoia[5] and Mira[15] were removed from the ranking due to decommissioning, while TOKI-SORA[2] and Summit[3] were newly ranked. TOKI-SORA consists of 5,760 nodes of PRIMEHPC FX1000, which has almost the same specification as Fugaku shown in Table 1, and our implementation is used for the evaluation.

Although omitted in Table 4, in June 2020, Fugaku achieved 70,980 GTEPS in $SCALE = 40$ using 92,160 nodes (60% of Fugaku) and also won the first place. Since this calculation scale is almost the same as the K computer, we will try to compare the two systems. The per-node performance of the K computer and Fugaku at 92,160 nodes is 377 MTEPS (31,302 GTEPS/82,944 nodes) and 770 MTEPS (70,980 GTEPS/92,160 nodes), respectively, so Fugaku has about twice the performance. As shown in Fig. 12, most of the communication time is occupied by collective communication (alltoallv and allgatherv), and the injection bandwidth is important for them. As described in Section 2, the difference in injection bandwidth between Fugaku and the K computer is a factor of two. Since the overall performance difference is also twice, we can assume that there is also a 2x difference in local calculation performance, but it is not as great as the specification. For example, the difference in bandwidth is 16.0 times. The reason why Fugaku's local performance is relatively low is that since the measurement is performed with 48 threads per process, there is a lot of memory access across CMGs. The performance modeling of BFS and the CMG-aware locality optimization of A64FX are the future works.

Green Graph500[1] is a ranking that evaluates the power efficiency performance (TEPS/W) among the systems ranked in Graph500. Green Graph500 is divided into two categories: the BIG DATA category deals with $SCALE \geq 30$, and the SMALL DATA category is for $SCALE \leq 29$. Since $SCALE = 30$ is a relatively small graph size, most top results in the BIG DATA category utilize only one node. Therefore, it can be said that the current Green Graph500 regulations are not suitable for a large-scale system such as Fugaku. As described in Section 6.1, BFS on Fugaku uses $SCALE = 41$ and Sequoia was the only machine that ran at the same size in Table 4. Since the power efficiency of Sequoia was 3.72 MTEPS/W, that of Fugaku is 1.9 times better than that of Sequoia.

7 Conclusion and future work

This paper presents the performance optimization of BFS in the Graph500 benchmark and evaluations conducted on Fugaku. In the performance evaluation using all Fugaku nodes for a large-scale graph consisting of about 2.2 trillion vertices and 35.2 trillion edges, we achieve 102,956 GTEPS and won the award in Graph500 in November 2020. This performance is 3.3 times that of the K computer, and 4.3 times that of Sunway TaihuLight which is the second place in the Graph500.

Future work will focus on the following: (1) We will optimize our BFS implementation to be aware of the four CMGs in A64FX. For this, NUMA architecture-aware techniques for BFS will be useful[21]. (2) Detailed performance modeling will be necessary to clarify the relationship between hardware and BFS performance. (3) We will develop various graph processing codes including SSSP in the Graph500 benchmark, and utilize Fugaku to perform graph processing of real-world data. (4) From the experiments in this paper, it was found that the communication time became dominant as the number of nodes increased. Future supercomputers for higher performance of BFS will require higher dimensional topologies than TofuD.

Acknowledgments

We would like to express our sincere thanks to Fujitsu engineers of the supercomputer Fugaku for helping us execute the benchmark. We are also grateful to Dr. Yutaka Ishikawa, the project leader of the Flagship 2020 Project. This work is partially funded by the Ministry of Education, Culture, Sports, Science and Technology (MEXT) program for the Development and Improvement for the Next Generation Ultra-High-Speed Computer System, under its Subsidies for Operating the Specific Advanced Large Research Facilities. This work is also partially funded by RIKEN Incentive Research Projects.

References

1. Graph500 and Green Graph500, https://graph500.org
2. Overview of JSS3, https://www.jss.jaxa.jp/en/
3. Summit, https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/
4. Andy Yoo et al: A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In: SC ’05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing. pp. 25–25 (2005). https://doi.org/10.1109/SC.2005.4
5. Barnes Peter D. et al: Warp speed: Executing time warp on 1,966,080 cores. In: Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. pp. 327–336 (2013)
6. Hiroyuki Miyazaki et al: Overview of the K computer. FUJITSU SCIENTIFIC and TECHNICAL JOURNAL **48**(3), 255–265 (2012)
7. Jordan Jakob et al: Extremely scalable spiking neuronal network simulation code: From laptops to exascale computers. Frontiers in Neuroinformatics **12**, 2 (2018)
8. Jure Leskovec et al: Kronecker graphs: An approach to modeling networks. Journal of Machine Learning Research **11**(33), 985–1042 (2010)
9. Koji Ueno et al: Efficient breadth-first search on massively parallel and distributed-memory machines. Data Science and Engineering **2**, 22–35 (2016)
10. Mitsuhisa Sato et al: Co-Design for A64FX Manycore Processor and ”Fugaku”. In: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 651–665. IEEE Computer Society, Los Alamitos, CA, USA (2020)
11. Peter Buhlmann et al (ed.): Handbook of Big Data. Chapman and Hall/CRC (2016). https://doi.org/10.1201/b19567
12. Richard C. Murphy, et al: Introducing the graph 500. In: Cray User’s Group (2010)
13. Scott Beamer et al: Direction-optimizing breadth-first search. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 12:1–12:10. SC ’12, IEEE Computer Society Press, Los Alamitos, CA, USA (2012)
14. Scott Beamer et al: Distributed memory breadth-first search revisited: Enabling bottom-up search. In: IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum. pp. 1618–1627 (2013)
15. Sean Wallace et al: Measuring Power Consumption on IBM Blue Gene/Q. In: 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum. pp. 853–859 (2013)
16. Sylvain Brohee, Jacques van Helden: Evaluation of clustering algorithms for protein-protein interaction networks. BMC Bioinformatics (2006), https://doi.org/10.1186/1471-2105-7-488
17. Yan-Fen Da, Xing-Ming Zhao: A Survey on the Computational Approaches to Identify Drug Targets in the Postgenomic Era. BioMed Research International pp. 1–9 (2015), http://dx.doi.org/10.1155/2015/239654
18. Yuetsu Kodama et al: Evaluation of Power Controls on Supercomputer Fugaku. In: Energy Efficient HPC State of the Practice Workshop in conjunction with IEEE Cluster2020. pp. 484–493 (2020)
19. Yuichiro Ajima et al: Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. Computer **42**(11), 36–40 (Nov 2009). https://doi.org/10.1109/MC.2009.370
20. Yuichiro Ajima et al: The Tofu Interconnect D. In: IEEE International Conference on Cluster Computing. pp. 646–654 (2018)
21. Yuichiro Yasui et al: Numa-optimized parallel breadth-first search on multicore single-node system. In: 2013 IEEE International Conference on Big Data. pp. 394–402 (2013)