Journal of New Media
DOI:10.32604/jnm.2021.018829
Article

BFS Parallel Algorithm Based on Sunway TaihuLight

Yang Zhou[1], Jinhui He[1] and Hao Yang[1,2,*]

[1]School of Computer Science, Chengdu University of Information Technology, Chengdu, 610225, China
[2]School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu, 610054, China
[*]Corresponding Author: Hao Yang. Email: vhyang@foxmail.com
Received: 22 March 2021; Accepted: 30 March 2021

Abstract: In recent years, more and more attention has been paid to the research and application of graph structure. As the most typical representative of graph structure algorithm, breadth first search algorithm is widely used in many fields. However, the performance of traditional serial breadth first search (BFS) algorithm is often very low in specific areas, especially in large-scale graph structure traversal. However, it is very common to deal with large-scale graph structure in scientific research. At the same time, the computing performance of supercomputer has also made great progress. China's self-developed supercomputer system Sunway TaihuLight (SW) has won the top 500 list for three consecutive times. The huge computing performance of supercomputer is the key to solve this problem. It can be seen that if we use the computing power of supercomputing to solve the problem of large-scale graph structure traversal, the efficiency of graph structure traversal will be greatly improved. This paper expounds how to realize the breadth first search algorithm of graph structure on the Sunway TaihuLight, and achieved some results. In this way, MPI and thread library called athread of SW platform are used, and the traversal performance is improved dozens of times through the above related technologies and some partition methods of graph structure.

Keywords: Sunway TaihuLight; breadth first search algorithm; parallel computing

1 Introduction

With the development and application of high-performance computer technology, it has become very difficult to improve the performance only by improving the frequency of single core. Parallel computing has become one of the key technologies to ensure the operational efficiency of large-scale computing applications. However, the parallelization of serial code is a very complicated task. Graph structure traversal algorithm, as a very common algorithm, has been used in many aspects, such as social network discovery, DNA sequence research, urban road planning and so on. However, the traditional serial graph structure traversal algorithm has low performance especially when dealing with some large graph structure data. How to parallelize the graph structure traversal algorithm is particularly important.

In this paper, according to some existing BFS parallel algorithm, and combined with Sunway TaihuLight platform to make corresponding improvements [1–4], so that BFS algorithm can run efficiently on Sunway TaihuLight platform.

2 Sunway TaihuLight Supercomputer Platform

Sunway TaihuLight is a supercomputer developed and installed in Wuxi National Supercomputing Center by china parallel computer engineering technology research center.

2.1 Sunway TaihuLight Architecture

Sunway TaihuLight is a supercomputer developed by china. The Sunway TaihuLight is designed as a supercomputer with heterogeneous architecture use SW26010 processor, which comprise of connected 40960 nodes, as show in Fig. 1, each node has one SW26010 processor [5].

The multi-core processor uses 64 bit autonomous Shenwei instruction system. The SW system with a peak performance of 12.54 GFLOPS, and a sustained performance of 9.3 GFLOPS.

2.2 Sunway Processor

The sw26010 processor contains four core groups (CGs). Core groups are connected by high-speed network on chip. The processor integrates 4 channels of memory and 8-channel pcie3.0 gigabit network card. Each core group contains a control core called management processing element (MPE) and a computing core array [6–8]. This array is an 8 × 8 array of 64 computing cores, called computing processing element (CPE) with a total of 260 cores.

Both the management processing element and computing processing element are 64-bit RISC (Reduced Instruction Set Computing) Instruction architecture. Both MPE and CPE have (Level 1) L1 instruction cached. MPE has L1 data cache and (Level 2) L2 data cache [9].

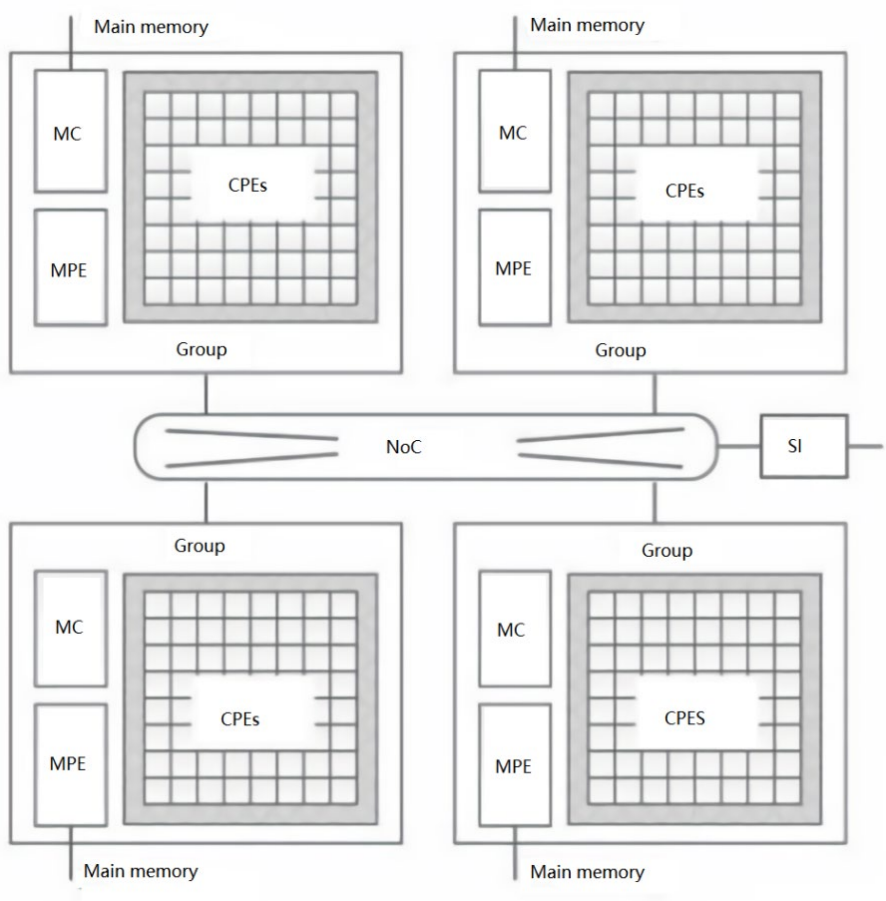

Figure 1: sw26010 processor

In order to use hardware resources efficiently, sw26010 processor's CPE does not have L1 or L2 data cache, instead, each computing processing element has a local data memory (LDM), the LDM's size is 64 Kb. Each CPE can access its own LDM or directly access main memory, but the delay of accessing LDM is lower (several clock cycles), and the delay of accessing main memory is higher (hundreds of clock

cycles). Also [10–12], CPE can use Direct Memory Access (DMA) to read or write a large amount of data to main memory at one time to reduce the pressure of memory access.

DMA supports reading or writing data asynchronously. Therefore, when using sw26010 programming, asynchronous operation can be used to cover the communication time with computing time. Because of the heterogeneity of sw26010 [13], the program needs to run on MPE and CPE at the same time. We can use openACC or athread to utilize the CPEs.

3 Problem Description

3.1 Breadth First Search

Breadth first search is a common graph search algorithm. Input the graph $G(V, E)$ composed of vertex set V and edge set E and search starting point S. BFS algorithm first accesses S, then travel the points $V_1, V_2, V_3, \ldots$ of S. Then according to V_1, V_2, V_3 sequence, start to accesses the adjacency points of these points set, the algorithm ends when all the vertices in the graph are traveled [14–16].

Algorithm 1: Serial BFS

```
for v in V
    π[v] ← –1
end for
π[s] ← s
F ← {s}
while F ≠ Ø
    G ← Ø
    for u connect to v
        π[v] = u
        insert(v) → G
    end for
    F ← G
end while
```

3.2 DataSet

The graph consists of two integers V, E and two 32-bit integer arrays v_pos and e_dst, The length of v_pos is $V + 1$, The length of e_dst is E, where the first integer in v_pos is 0，the integer value of index v is e. For a certain vertex i, we take the two values e_1 and e_2 in the array v_pos with indexes i and $i + 1$. Then, in the e_dst array, all the values from index e_1 to e_2 are the vertices connected by vertex i.

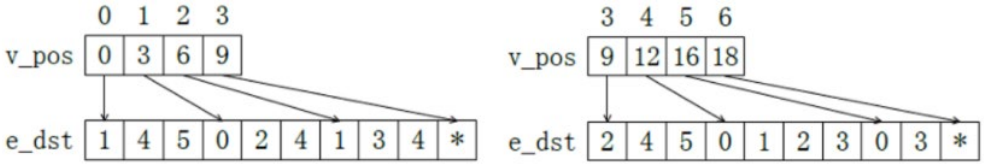

Figure 2: Storage of graph structure

4 Parallelization

4.1 Parallel By MPI

At present, there are many successful researches on BFS algorithm parallelization, and inter layer

synchronous parallel is a common one [17]. This algorithm needs an upper queue and a lower queue. It looks for the vertices connected with the vertices in the upper queue, processes the state information, and adds them to the lower queue. After that, the vertices in the lower queue are put into the upper queue and the lower queue is cleared. After these operations, all processes need to be synchronized.

In this algorithm, graph data G, visited state array and spanning tree are shared by multiple threads. During the search process, the thread will search for the vertices that have been accessed by other threads in the layer search, thus resulting in invalid search [18]. In Fig. 3, the thick line represents an invalid search because the vertex it refers to has been accessed by thread 1. This classic BFS algorithm starts from the visited vertices to find the unreachable vertices, which is called top-down search. After parallelization, the invalid search problem caused by repeated discovery has a great impact on the search performance.

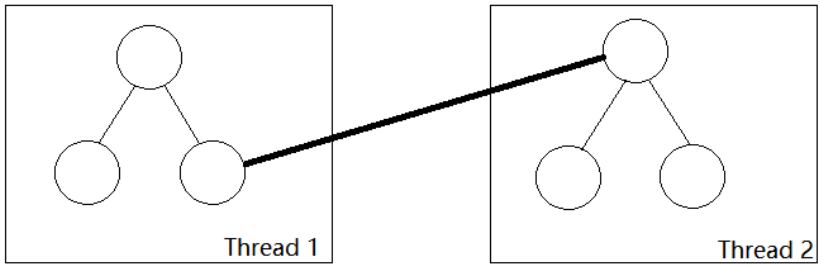

Figure 3: Invalid search

Because of the storage method of graph structure, we can divide graph by vertex. In the multi-process parallel environment using MPI, each process divides a part of the vertices, and each process only maintains the information of the vertices which contains [19–21]. When the vertices that other processes need (traversed) are not in the process, MPI communication is needed to exchange information.

For example, there is an undirected graph, $G = \{< 0,2 >, < 1,2 >, < 1,3 >\}$. We use two processes to traverse the graph in parallel. Process 0 has two vertices, V_0 and V_1, and process 1 has two vertices, V_2 and V_3(Fig. 4). Among them, the circle indicates that vertex assigned by the process, and the state information of the vertex needs to be maintained. The rectangle indicates that the process does not contain this vertex, but will be connected to this vertex. In the process of graph parallel traversal, when a process traverses a circle vertex, the vertex state information can be directly processed by this process [22]. If a rectangle vertex is traversed, it is necessary to find which process the vertex belongs to, then communicate with that process, and then process the state information of the vertex.

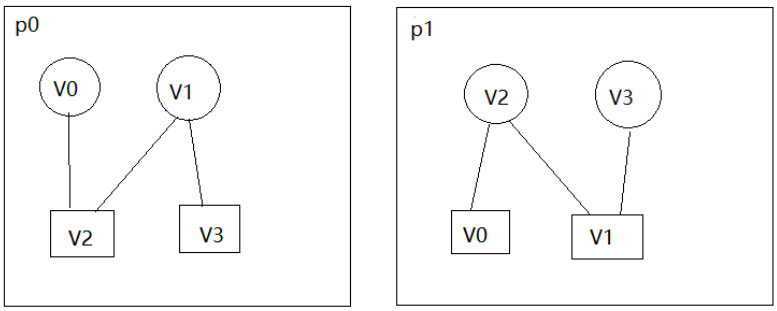

Figure 4: Graph divide

In this algorithm, we need to find the vertex belongs to which process according to the partition of the current graph, each process needs a group of queues to store the traversed vertices. The number of queues is the same as the number of processes. After all vertices of the same layer of all processes are

processed, the vertex data in the corresponding queue is sent to the corresponding process through the all to all operation of MPI [23–24]. In this way, each process only receives its own vertex information. Then, the received vertex state is processed, and then it is added to the upper queue without repetition, and all processes start the next round of traversal.

Algorithm 2: A round operation of one process

```
SendQueue[S] ← Ø
RecvQueue[S] ← Ø
for v in FS
    u is connect to v
    p = findOwner(u)
    SendQueue[p] = u
end for
FS ← Ø
all to all send SendQueue to RecvQueue
merge RecvQueue to FS
```

4.2 Parallel by Athread

There are 64 computing cores in every CG. How to use these cores efficiently is the key to improve the efficiency of parallel BFS. Based on the previous parallel optimization of BFS using MPI, since one core group obtains some vertices of the graph, when searching the adjacent vertices of each vertex in upper queue, the search task should be assigned to multiple computing cores, that is, each computing core is responsible for the task of finding adjacent vertices for some vertices in the upper queue.

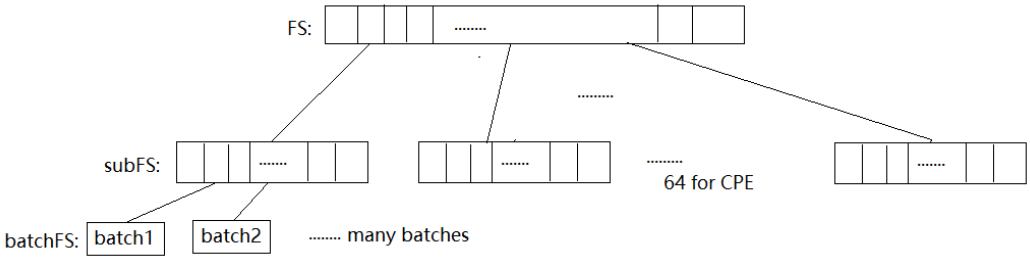

Figure 5: FS queue divide

Because there is no high-speed data cache in the computing core, instead, a 64KB LDM is used. For some larger graphs, the number of vertices in the upper queue is very large for each process. Even if it is divided into 64 copies(subFS), the amount of data will exceed 64KB. Therefore, when using LDM, it is necessary to divide v_pos into many batches, and copy batch data(batchFS) into LDM. Therefore, two buffers are needed on each computing core to store batchFS and e_dst data. In addition, due to the limited space of LDM, a queue is designed in each computing core to store the vertex data of the lower queue(FN). This queue stores tuple metadata (pid, vertex), where pid represents the number of the process to which the vertex belongs, and vertex is the vertex information.

How to use the slave processor efficiently will directly affect the acceleration effect of the algorithm. Because the local storage space of each slave processor is too small compared with a large

number of graph structure data, in the whole operation process, the local storage space of the slave processor will frequently exchange data with the main memory, that is, the main core frequently uses DMA process to copy batch data.

For a core group, it has one control core and 64 computing cores. Each core contains a buffer queue in its LDM. When this queue is written back to main memory, there will be write conflicts. Because Sunway platform does not provide mutex operation on main memory, using semaphore will cause frequent thread hang and wake-up, which seriously affects efficiency. Therefore, we design 64 cache queues on the master to store the data written back by each computing core, and then the main core merge the data in this cache queue into the SendQueue according to the pid value. It is worth noting that the operation of merge the cache queue into SendQueue and traversing on computing core are performed simultaneously.

Algorithm 3: Merge operation on master

```
1:  while CPEs are alive
2:    for f in FN_i
3:      for (pid, v) in f
4:        append v to SendQueue[pid]
5:      end for
6:      clean f
7:    end for
8:  end while
```

Algorithm 4: Parallel search on each computing core

```
1:  subFS_ldm ← Ø
2:  FN_ldm ← Ø
3:  for batchFS in subFS
4:    copy batchFS into batchFS_ldm
5:    for i in batchFS_ldm
6:      pA = v_pos[i], pB = v_pos[i + 1]
7:      copy e_dst[pA] to e_dst[pB] into e_dst_ldm
8:      for v in d_dst_ldm
9:      p = findOwner(v)
10:     append (p, v) to FN_ldm
11:     if FN_ldm is full
12:       copy FN_ldm to FN_i
13:     end if
14:   end for
15: end for
```

Note that the operation in Algorithm 4 line 12 and Algorithm 4 line 2 to line 7 represents the need for mutual exclusion between the main core and the computing core, which can be implemented by

semaphores in actual code. The purpose of designing multiple FN queues is to avoid mutual exclusion between computing cores, and the performance loss caused by mutual exclusion between single computing core and main core is within acceptable range.

For the 6 line of Algorithm 3, computing core directly accessing v_pos on main memory with high delay, in addition, copying the value of e_dst repeatedly can seriously affect performance. Therefore, we can sort the values in batchFS_ldm. Then, when accessing v_pos, it will have great spatial locality. At the same time, the access to e_dst is also sequential.

Therefore, this algorithm can be improved. The key to improve the performance is how to access the memory efficiently. Because the local memory of sw26010 is too small, and the function calls from the core will consume the stack memory, the memory left for storing the graph structure data is smaller. This small local storage space can be used as the cache in modern CPU (there is no data cache in sw26010 processor). Therefore, in order to improve the hit rate of the manually implemented cache, we need to sort the data to make the data have good spatial locality.

Combined with the characteristics of Sunway platform, we can manually create a cache to store v_ pos and e_ dst data. The improved algorithm is as follows:

Algorithm 5: parallel search on each computing core(improved)

```
1:  subFS_ldm ← Ø
2:  FN_ldm ← Ø
3:  for batchFS in subFS
4:     copy batchFS into batchFS_ldm
5:     sort batchFS_ldm
6:     for i in batchFS_ldm
7:        pA = cacheRead(v_pos[i]), pB =cacheRead(v_pos[i + 1])
8:        if pB To pB not in e_dst_ldm
9:           copy e_dst[pA] to e_dst[pB] into e_dst_ldm
10:       end if
11:       for v in d_dst_ldm
12:          p = findOwner(v)
13:          append (p, v) to FN_ldm
14:          if FN_ldm is full
15:          copy FN_ldm to FN_i
16:       end if
17:    end for
18: end for
```

5 Conclusion

We selected four typical data sets to test the performance of the algorithm. These four data sets cover protein related data sets, and road data sets in USA and Europe. The ratio of vertex number and pass of these data sets are different. In this way, the performance of parallel graph traversal algorithm to deal with different structure graph data is tested. The results show that, this paper presents the performance of parallel graph traversal algorithm in response to different structure graph data the parallel graph traversal

algorithm shown in the paper can achieve a good acceleration effect when dealing with various graph structures. In comparison, the acceleration effect of cage15 data set is satisfactory. For the data set with large graph scale, such as eurpos_osm, when the number of cores exceeds a certain extent, the acceleration effect increases slightly.

In this optimization result test, a total of 4 examples are used for testing, and the information of vertex and edge of examples are as follows:

Table 1: Use case

Use cases	Vertices	Edges
Freescale1	3428755	16945664
cage15	5154859	94044692
road_usa	23947347	57708624
Europe_osm	50912018	108109320

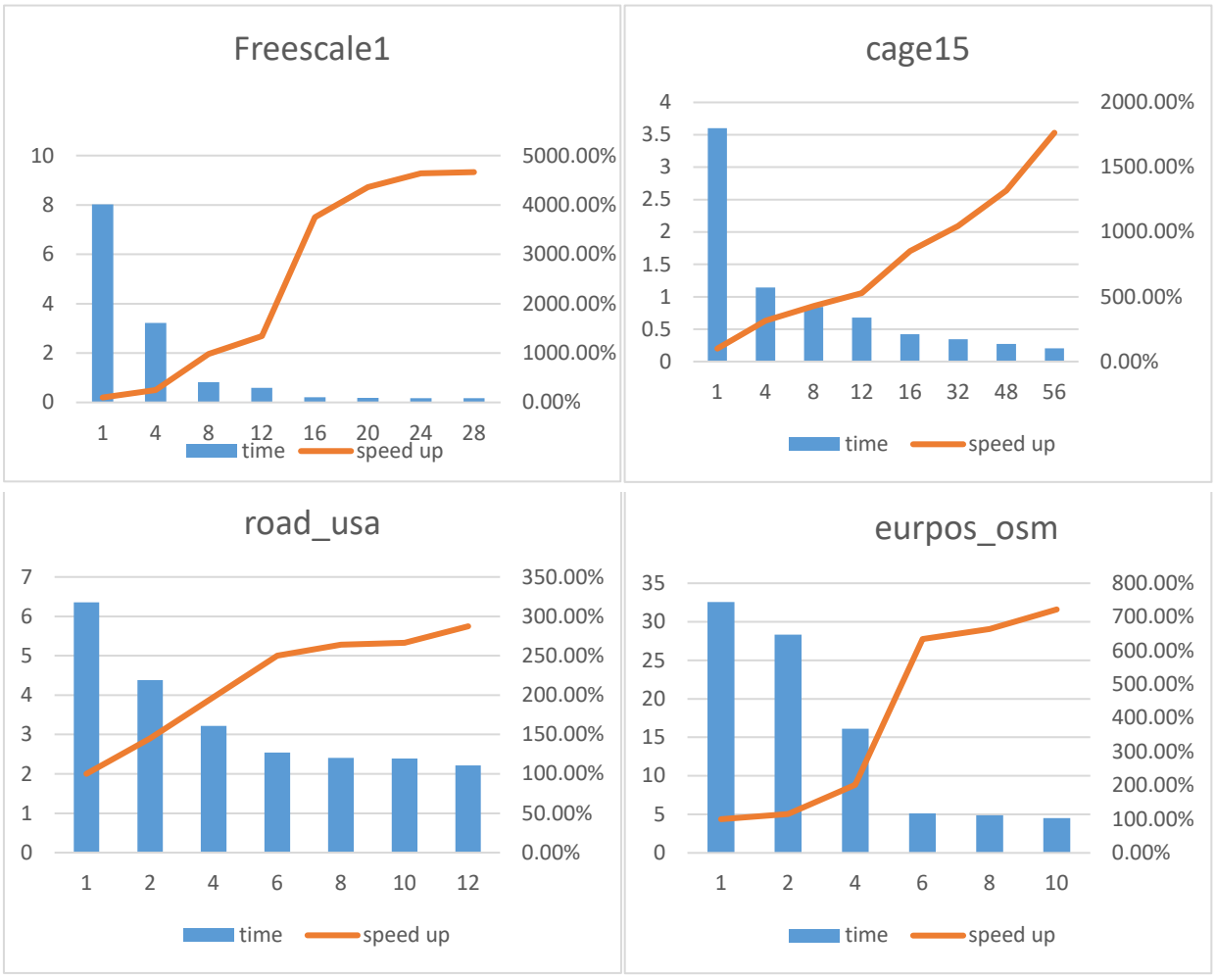

Figure 6: Conclusion

According to the above performance record table, the graph structure parallel traversal algorithm for Sunway TaihuLight platform described in this paper has achieved certain optimization effect for four data. For the parallel optimization methods described in this paper, some problems still need to be solved, such as evenly distributing the FS queue according to the number of computing cores, and for some side nodes

in the graph (that is, the vertex is connected to many vertices) In this case, e_ dst_ldm queue will become large, and it is prone to the problems of insufficient LDM space and unbalanced load.

Funding Statement: This work is sponsored by the Sichuan Science and Technology Program (2020YFS0355 and 2020YFG0479).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] W. J. Hong, K. L. Li, Z. Quan, W. D. Yang, K. Q. Li *et al.,* "PETSc's heterogeneous parallel algorithm design and performance optimization on the Sunway Taihulight system", *Chinese Journal of Computers*, vol. 40, no. 9, pp. 2057–2069, 2017.

[2] L. J. Jiang, C. Yang, Y. L. Ao, W. W. Yin, W. J. Ma *et al.,* "Towards highly efficient DGEMM on the emerging SW26010 many-core processor", in *2017 46th Int. Conf. Parallel Processing,* Bristol, UK, pp. 422–431, 2017.

[3] Y. L. Ao, C. Yang, F. F. Liu, W. W. Yin, L. J. Jiang *et al.,* "Performance optimization of the HPCG benchmark on the Sunway Taihulight supercomputer", *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 1, pp. 11, 2018.

[4] Y. Yu, H. An, J. S. Chen, W. H. Liang, Q. Q. Xu *et al.,* "Pipelining computation and optimization strategies for scaling GROMACS on the Sunway many-core processor", in *Proc. 17th Int. Conf. Algorithms and Architectures for Parallel Processing*, Helsinki, Finland, pp. 18–32, 2017.

[5] S. M. Chen, L. Peng, S. Irving, Z. Zhao, W. H. Zhang *et al.,* "qSwitch: Dynamical off-chip bandwidth allocation between local and remote accesses", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 75–87, 2018.

[6] W. H. Zhang, X. F. Ji, B. Song, S. Q. Yu, H. B. Chen *et al.,* "VarCatcher: A framework for tackling performance variability of parallel workloads on multi-core", *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1215–1228, 2017.

[7] R. E. Tarjan, "Depth-first search and linear graph algorithms", *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[8] W. McLendon III, B. Hendrickson, S. J. Plimpton and L. Rauchwerger, "Finding strongly connected components in distributed graphs", *Journal Parallel Distributed Computing*, vol. 65, no. 8, pp. 901–910, 2005.

[9] S. Beamer, K. Asanović and D. A. Patterson, "Direction-optimizing breadth-first search", in *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2012.

[10] A. Buluc and J. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[11] J. D. Ullman and M. Yannakakis, "High-probability parallel transitiveclosure algorithms," *SIAM Journal on Computing*, vol. 20, no. 1, pp. 100–125, 1991.

[12] C. Leiserson and T. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Sym. on Parallelism in Algorithms and Architectures*, pp. 303–314, 2010.

[13] Y. Xia and V. Prasanna, "Topologically adaptive parallel breadth-first search on multicore processors," in *Int. Conf. on Parallel and Distributed Computing Systems (PDCS)*, 2009.

[14] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Int. Conf.e on High-Performance Computing (HIPC)*, 2007, pp. 197–208.

[15] S. Hong, S. Kim, T. Oguntebi and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *Sym. on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[16] D. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *Proc. 35th Int. Conf. on Parallel Processing (ICPP)*, pp. 523–530, 2006.

[17] R. Pearce, M. Gokhale and N. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11, 2010.

[18] S. Beamer, A. Buluc, K. Asanovic and D. Patterson, "Distributed memory breadth-first search revisited: Enabling bottom-up search," in *Proc. of IEEE 27th Int. Parallel and Distributed Processing Sym. Workshops and PhD Forum*, pp. 1618–1627, 2013.

[19] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *Proc. of the Int. Parallel and Distributed Processing Sym.*, 2010, pp. 425–434, 2014.

[20] A. Buluc and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proc. of Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2011.

[21] H. Liu and H. H. Huang, "Enterprise: Breadth-first Graph Traversal on GPUs," in *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, pp. 68:1—-68:12, 2015.

[22] K. Ueno and T. Suzumura, "Parallel distributed breadth first search on GPU," in *Proc. of 20th Annual Int. Conf. on High Performance Computing*, pp. 314–323, 2013.

[23] M. Bisson, M. Bernaschi and E. Mastrostefano, "Parallel distributed breadth first search on the kepler architecture," in *Proc. of IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2091–2102, 2016.

[24] D. Merrill, M. Garland and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, vol. 47, no. 8, pp. 117, 2012.