Scalable Graph Traversal on Sunway TaihuLight with Ten Million Cores

Heng Lin†, Xiongchao Tang†, Bowen Yu†, Youwei Zhuo[1,‡],
Wenguang Chen†§, Jidong Zhai†, Wanwang Yin§, Weimin Zheng†
†*Tsinghua University,* ‡*University of Southern California,*
§*Research Institute of Tsinghua University in Shenzhen,*
¶*National Research Centre of Parallel Computer Engineering and Technology*
Email: {*linheng11, txc13, yubw15*}*@mails.tsinghua.edu.cn,*
{*cwg, zhaijidong, zwm-dcs*}*@tsinghua.edu.cn, yinwanwang@gmail.com*

***Abstract*—Interest has recently grown in efficiently analyzing unstructured data such as social network graphs and protein structures. A fundamental graph algorithm for doing such task is the Breadth-First Search (BFS) algorithm, the foundation for many other important graph algorithms such as calculating the shortest path or finding the maximum flow in graphs.**

In this paper, we share our experience of designing and implementing the BFS algorithm on Sunway TaihuLight, a newly released machine with 40,960 nodes and 10.6 million accelerator cores. It tops the Top500 list of June 2016 with a 93.01 petaflops Linpack performance [1].

Designed for extremely large-scale computation and power efficiency, processors on Sunway TaihuLight employ a unique heterogeneous many-core architecture and memory hierarchy. With its extremely large size, the machine provides both opportunities and challenges for implementing high-performance irregular algorithms, such as BFS.

We propose several techniques, including pipelined module mapping, contention-free data shuffling, and group-based message batching, to address the challenges of efficiently utilizing the features of this large scale heterogeneous machine. We ultimately achieved 23755.7 giga-traversed edges per second (GTEPS), which is the best among heterogeneous machines and the second overall in the Graph500s June 2016 list [2].

***Index Terms*—Breadth-First Search; heterogeneous; parallel algorithm; micro-architecture;**

1. Introduction

Recently there has been a burst of interest in analyzing unstructured data, such as social network graphs or protein structures. However, applications that analyze such data often suffer from frequent random memory accesses and load imbalances. A fundamental graph algorithm in analyzing unstructured data is the Breadth-First Search (BFS) algorithm, the basis for many other important graph algorithms, such as the Shortest Path algorithm, or the Maximum Flow algorithm. BFS has been selected as the Graph500 benchmark to measure the computing power of super-computing systems for irregular and data-intensive applications.

[1] *Youwei took part in this work at Tsinghua University.*

Although considerable amount of research has been devoted to implementing BFS on large-scale distributed systems [3]–[5], rather less attention has been paid to systems with accelerators.

Notably, only one machine among the top ten in the Graph500 list of November 2015 (Tianhe-2) has accelerators [2]. Furthermore, Tianhe-2 does not use the accelerators for its Graph500 benchmark test, while the accelerators are fully employed for its Linpack performance.

The question of how irregular applications, such as BFS, can leverage the power of accelerators in large-scale parallel computers is still left to be answered.

In this paper, we share our experience of designing and implementing the BFS algorithm on Sunway TaihuLight, a newly released machine with 40,960 nodes or 10.6 million accelerator cores. It tops the Top500 list of June 2016 with a 93.01 petaflops (PFlops) Linpack performance [1].

Designed for extremely large-scale calculations and cost/power-effectiveness, the processors in the Sunway TaihuLight system employ a unique heterogeneous many-core architecture and memory hierarchy. Every processor in TaihuLight contains four general-purpose cores, each of which attaches 64 on-chip accelerator cores. Each accelerator core has a 64 KB on-chip scratch pad memory. The four general-purpose cores and the 256 accelerator cores can both access 32GB shared off-chip main memory. Because it is generally difficult and expensive in chip design and manufacturing to ensure cache coherence among cores, TaihuLight chooses not to offer the cache coherence in exchange for more chip space for computing. That heterogeneous architecture provides both opportunities and challenges implementing irregular algorithms, such as BFS. The following features of TaihuLight significantly impact the BFS design:

Large overhead for thread-switching Neither the general purpose cores nor the accelerator cores have corresponding hardware that supports multi-threading. Without efficient multi-threading support, it is difficult to implement BFS, which needs asynchronous communications.

Limited communication patterns for accelerator cores The 64 accelerator cores attached to every general purpose core only have an 8×8 mesh layout

for on-chip communication. Communications are only allowed between accelerator cores in the same row or column. Deadlock-free communications for any arbitrary pair of accelerator cores are not supported, while it is difficult to restrict the arbitrary communication patterns for irregular algorithms on accelerator cores.

Large scale problem with over-subscribed interconnect The TaihuLight has 40,960 nodes that are connected by an FDR InfiniBand (56 Gbps) with a two-level fat tree topology. The lower level connects 256 nodes with full bandwidth, while the upper level network connects the 160 lower level switches with a 1:4 over-subscription ratio. In a simple BFS algorithm, there are at least one message transfer, which is termination indicator of current transmission, for each pair of nodes. What's more, if the graph is a power-law graph, most of the vertices' degree are small, there will be lots of very small messages. Such small messages make very inefficient use of bandwidth, causing poor communication performance.

We propose the following techniques:

Pipelined modules mapping To use the single-threaded general-purpose core for asynchronous communications, we leverage the fact that each CPU contains four general cores, sharing a physical memory. We decompose the asynchronous BFS algorithm to several steps, such as receiving messages, sending messages, processing edges, and forwarding messages, and we assign each of these steps to one general-purpose core (network communication and scheduling work) or the accelerator cores (most of the computing work). By carefully arranging the memory space and the read/write sequences, we can make our code run correctly and efficiently without requiring cache coherence for shared memory.

Contention-free data shuffling The accelerator cores are mainly used for edge processing and generating messages to remote nodes, which have to be grouped for batched communications. The key step of edge processing is data shuffling, i.e., generating a message for each edge to the computing node, which contains the destination vertex. To overcome the limited communication patterns for accelerator cores and avoid contentions in writing into the main memory, we propose a contention-free data shuffling schema for accelerator cores by dividing the accelerator cores into three groups: consumers, routers, and producers. The router cores are used to route message from producers to consumers in a deadlock-free way, while consumer cores write to non-overlapped main memory regions to avoid contention.

Group-based message batching To reduce the number of small messages, we propose a two-level message passing approach. Nodes are divided into groups. All messages from one node to another group are batched into a large message and sent to the relay node in the destination group. The received message is then decomposed, and each original message is forwarded to its real destination in the group by the relay node. Although this approach seems to slightly increase the number of messages, it benefits from the fact that most messages are transferred inside a group, which is connected by lower level switches that provide full bandwidth.

We ultimately achieved 23755.7 giga-traversed edges per second (GTEPS) following the specifications of the Graph500 benchmark, which is the best result for heterogeneous machines and second overall in the June 2016 Graph500 list.

The remainder of this paper is organized as follows. We will introduce the BFS algorithm in Section 2 and the architecture of TaihuLight in Section 3. Section 4 and Section 5 describe our methodology and implementations, respectively. After evaluating our work in Section 6, we present related work in Section 7. Section 8 provides a brief discussion and Section 9 concludes the paper.

2. Breadth-First Search

In this section, we present the basic parallel Breadth-First Search (BFS) algorithm that will be optimized on the TaihuLight system. Further, we split the BFS algorithm into several modules and analyze the main characterizations.

2.1. Parallel BFS Algorithm

Algorithm 1: Parallel hybrid BFS framework

Input:
$G = (V, E)$: graph representation using CSR format
v_r: root vertex
Output:
Prt: parent map
Data:
$Curr$: vertices in the current level
$Next$: vertices in the next level
$state \in \{$*topdown*, *bottomup*$\}$: traversal policy

```
1  Prt(:) ← −1, Prt(v_r) ← v_r
2  Curr ← ∅
3  Next ← {v_r}
4  while Next ≠ ∅ do
5      Curr ← Next
6      Next ← ∅
7      state ← TRAVERSAL_POLICY()
8      if state = topdown then
9          Fork FORWARD_HANDLER thread
10         FORWARD_GENERATOR()
11     if state = bottomup then
12         Fork FORWARD_HANDLER thread
13         Fork BACKWARD_HANDLER thread
14         BACKWARD_GENERATOR()
15     Join threads and barrier
```

In this work, we select a 1-dimensional (1D) partitioning with a direction-optimized algorithm [4], [6] as our basic framework for the following two major reasons.

First, as the degree of the input graph follows a power law distribution, the direction optimization [7] can skip massive unnecessary edge look-ups by combining the conventional Top-Down traversal with a Bottom-Up traversal. As shown in Algorithm 1, there is a TRAVERSAL_POLICY

Algorithm 2: Parallel hybrid BFS: helper functions

Input:
$G = (V, E)$: graph using CSR format
$Curr$: vertices in the current level
$Next$: vertices in the next level
Output:
Prt: parent map

```
1  Function FORWARD_GENERATOR()
2  for u ∈ Curr do
3      for v : (u, v) ∈ E do
4          SEND_FORWARD (u, v) to OWNER(v)

5  Function BACKWARD_GENERATOR()
6  for v ∈ V and Prt(v) = −1 do
7      for u : (u, v) ∈ E do
8          SEND_BACKWARD (u, v) to OWNER(u)

9  Function FORWARD_HANDLER()
10 RECEIVE_FORWARD (u, v)
11 if Prt(v) = −1 then
12     Prt(v) ← u
13     Next ← Next ∪ v

14 Function BACKWARD_HANDLER()
15 RECEIVE_BACKWARD (u, v)
16 if u ∈ Curr then
17     SEND_FORWARD (u, v) to OWNER(v)
```

function that utilizes runtime statistics to heuristically determine whether Top-Down or Bottom-Up is the more efficient traversal strategy. The details of the heuristic traversal policy can be found in a previous work [7].

Second, the 1D partitioning method uses the Compressed Sparse Row (CSR) data structure to partition the adjacency matrix of the input graph by rows. Therefore, each vertex of the input graph belongs to only one partition. We implement this method in an asynchronous way to not only eliminate barrier operations but also allow for the simultaneous execution of different computational modules within the same process in BFS.

Algorithm 2 elaborates on the main functions of Algorithm 1, with the computation part of these functions divided into the dispose module and the reaction module, as shown in Figure 1. These are the most time-consuming parts of BFS. Section 4.4 discusses the *Forward Relay* and *Backward Relay* modules. Other modules can be mapped to the same-named functions in Algorithm 2. For example, the *Forward Generator* module is mapped to the FORWARD_GENERATOR function. Some of these modules are shared between Top-Down and Bottom-Up traversals.

The reaction module consists of four steps: reading data from memory, processing the data in the computing unit(s), writing data to memory, and sending the data to other nodes. The last two steps are sometimes omitted according to the result of prior steps. For example, *Forward Generator* will read the frontier and graph data from memory. If data exist that need to be sent to other nodes, the data will be written into a corresponding send buffer, and the data will be sent to the corresponding nodes. Other reactions like *Backward Generator* have a similar operation pattern and mainly differ in the second step of processing data in the computing unit(s).

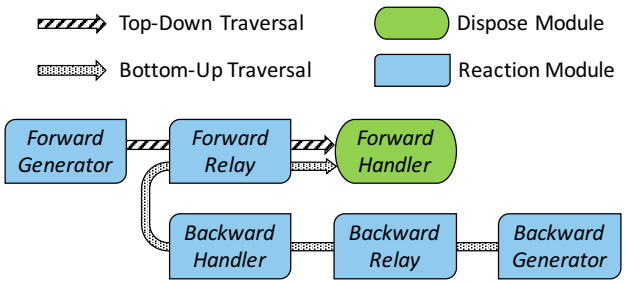

Figure 1: Modules in Top-Down and Bottom-Up processes of BFS. We have two types of modules—reaction and dispose. Only the reaction modules send the processed data to other nodes. In this case, *Forward Handler* is a disposed module and all others are reaction modules. *Forward Relay* and *Backward Relay* modules are discussed in Section 4.4.

Comparing it to the reaction module, the dispose module is simpler, as it only contains the preceding three steps of the reaction module and doesn't produce any data to send. For example, *Forward Handler* reads the messages from other nodes, checks the frontier, and updates the result in main memory.

2.2. Characterization of BFS

BFS is a typical graph application with four main characteristics, which are shared by most graph applications.

Frequent data access As BFS contains very few computing operations, so it is hard to make use of vector operations. Within one computing node, memory access speed is the main performance bottleneck, while the network communication speed is the bottleneck in a distributed system.

Random data access As the neighbors of a vertex can be any other vertices in the input graph, traversing the graph requires accessing many parts of the data in a random manner that cannot be predicted beforehand. In an asynchronous BFS framework, the random access becomes random main memory accesses, ranging from several megabytes to several hundred megabytes, and random messages between any pair of nodes.

Data dependence Memory access patterns of BFS are highly input-dependent, which means memory access locations cannot be determined until the data is read into the memory. It is therefore difficult to use classic optimizations such as prefetch to improve the performance of memory access.

Non-uniform data distribution The vertices in input graphs usually follow a power law distribution, and the distribution of the degree of each vertex is very imbalanced. Achieving the optimal performance is challenging because imbalanced vertex degrees cause significant load balance.

2.3. Graph500

Complementary to the Top500 [1], Graph500 [2], first announced in November 2010, is a new benchmark for large-

Item	Specifications
MPE	1.45 GHz, 32KB L1 D-Cache, 256KB L2
CPE	1.45 GHz, 64KB SPM
CG	1 MPE + 64 CPEs + 1 MC
Node	1 CPU (4 CGs) + 4×8GB DDR3 Memory
Super Node	256 Nodes, FDR 56 Gbps Infiniband
Cabinet	4 Super Nodes
TaihuLight	40 Cabinets

TABLE 1: Sunway TaihuLight specifications.

scale data analysis problems that has received wide acceptance. The ranking kernel of Graph500 is BFS along with a set of specifications.

In detail, the Graph500 benchmark consists of the following steps: (1) generate raw input graph (edge list, etc.), (2) randomly select 64 nontrivial search roots, (3) construct graph data structures, (4) run BFS kernel using the 64 search roots, (5) validate the result of BFS, and (6) compute and output the final performance of BFS.

We focus on the BFS kernel step because other steps should be scalable in large parallel environments.

3. TaihuLight Architecture and Challenge

Unlike other heterogeneous supercomputers that use standalone many-core accelerators (e.g., Nvidia GPU or Intel Xeon Phi), TaihuLight uses an integrated SW26010 CPU consisting of 260 cores in each node. We introduce the computing and network subsystems of TaihuLight here, while other details can be found in [8], [9]. Table 1 lists some important specifications of TaihuLight. We will elaborate on these specification numbers in the following subsections.

3.1. SW26010 CPU

Each node of the TaihuLight has one SW26010 CPU, which is designed as a highly integrated heterogeneous architecture for high performance computing. Illustrated in Figure 2, each CPU has four core groups (CGs), connected by a low latency on-chip network. Each CG consists of one management processing element (MPE), one computing processing element (CPE) cluster, and one memory controller (MC).

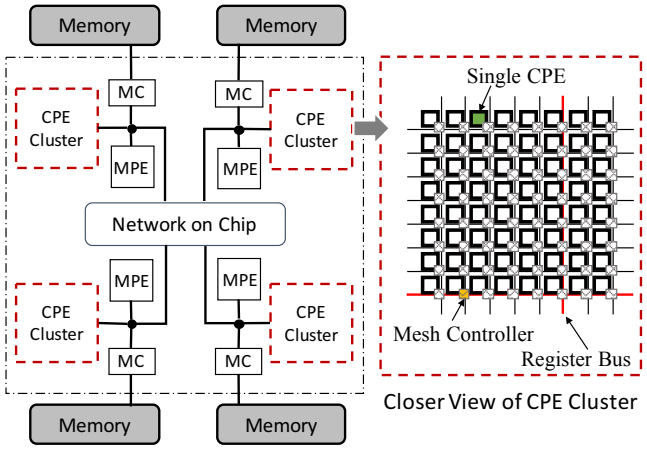

Figure 2: CPU Architecture of TaihuLight

The MPE is a fully functional 64-bit RISC general purpose core, which supports superscalar processing, memory management, and the interrupt function. Nevertheless, some features have limited performance. For instance, the latency of system interrupt is about 10 us, which is ten times that of Intel CPU's, and MPEs within one CPU do not support cache coherence. Each MPE has a 32 KB L1 data cache, a 32 KB L1 instruction cache, and a 256 KB L2 cache. Both MPEs and CPEs work at a frequency of 1.45 GHz.

The CPE is a simplified architecture of 64-bit RISC accelerator core. To minimize the complexity of the CPE, some unnecessary functions were removed, such as system interrupt. Further, CPEs only support atomic increase for atomic operations in main memory. Each CPE has a 16 KB L1 instruction cache and a 64KB scratch pad memory (SPM). Similar to shared memory in typical GPUs, programmers need to explicitly control the SPM.

64 CPEs attaching to the same MPE are organized as a CPE cluster. CPEs within a cluster are connected in an 8*8 mesh topology by register buses. CPEs in the same row or column can communicate to each other using a fast register communication, which has very low communication latency. In one cycle, the register communication can support up to 256-bit communication between two CPEs in the same row or column, and there are no bandwidth conflicts between different register communication of CPEs.

The different features of MPE and CPE determine the types of assignable work. We summarize some of the challenges involved in applying BFS to SW26010 CPU.

- Asynchronous BFS framework requires several modules running simultaneously, which the MPE supports poorly because it can only efficiently support a single thread. Further, there is no shared cache among MPEs, which means MPEs have to use main memory for communication, whose latency is around one hundred cycles, thus exacerbating the issue.
- In terms of notifications among the CPEs and MPEs, the large overhead of system interrupt makes it impractical; thus, we use the global memory for higher latency interaction.
- BFS accesses a large range of data, normally several MB, randomly. However, the SPM size of each CPE is only 64 KB. In the memory hierarchy, the next level of SPM is global memory, which has a latency that is 100 times larger. Collaboratively using the whole SPM in a CPE cluster is a possible solution.
- Collaboration within CPEs can use either the main memory or the register communication. However, the incomplete atomic operation significantly limits the former usage. It is inefficient to only use the atomic increase operation to implement other atomic functions such as compare-and-swap.
- The register communication between CPEs is supported by a mesh network using synchronous explicit messaging. The random access nature of BFS makes it easy to cause a deadlock in the register communication once the messaging route includes a cycle.

3.2. Main Memory

In Section 3.1, we showed that each CG has one memory controller, connected to one 8 GB DDR3 DRAM. Therefore, each node has 32 GB memory in total (4 DRAMs). We configure most of this memory for cross segmentation, which means that the memory is shared among all the MPEs and CPEs within a CPU that have the same bandwidth.

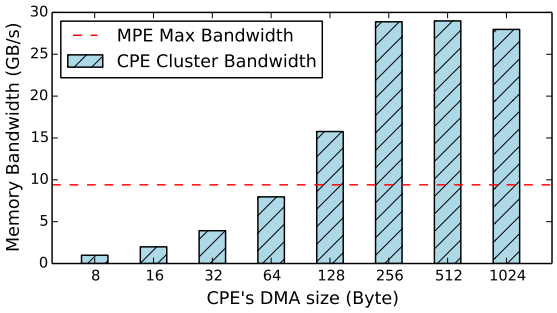

Figure 3: Direct memory access (DMA) bandwidth benchmark of a CPE cluster with different chunk sizes. A CPE cluster can get the desired bandwidth with a chunk size equal to or large than 256 Bytes. Compared to that of MPE, the speed CPE clusters accessing the memory is 10 times faster than the MPE. Although speed is measured by read operations, while write has a similar performance.

Since BFS is a memory-intensive application, how to utilize the memory bandwidth efficiently is a huge concern. MPEs do not support efficient multi-thread programming, thus one thread per MPE is the only practical implementation. When we set the batch size of memory accesses to 256 bytes for every group, the maximum memory bandwidth MPEs can achieve is 9.4 GB/s. However, CPE clusters can achieve a much higher bandwidth as high as 28.9 GB/s, as shown in Figure 3. In BFS design, we should make use of CPE clusters for most operations.

3.3. Network

The Network topology of TaihuLight is a 2-level fat tree, including a super node network at the bottom and a central switching network at the top. The network uses a static destination-based strategy for its route policy.

Each super node has 256 nodes connected by high-bandwidth and low-latency network. Full bisection bandwidth network is provided within each super node. Central switching network is responsible for communication among super nodes, which is designed to use only a quarter of the potential bandwidth instead of a fully connected network. TaihuLight uses FDR 56Gbps network interface cards and provides a 70TB/s bisection network bandwidth in total.

Fat-tree-based networks are widely used in large scale supercomputers, the random access nature of BFS poses the following challenges for the network in TaihuLight.

- The random access nature of BFS algorithm produces messages between each pair of nodes. In reality, every connection uses 100 KB memory due to the MPI library, so an MPE needs 4 GB memory just for establishing connections, which is impractical for memory intensive applications like BFS.
- Most of the messages between nodes are quite small, and managing such a large number of small messages efficiently under a customized fat tree is challenging.

4. Methodology

4.1. Principle

We discussed the BFS algorithm in Section 2 and the TaihuLight architecture in Section 3. The complex architecture of TaihuLight and the irregular characterization of the BFS algorithm pose several challenges to the design of the optimal BFS on such a large-scale heterogeneous system. We list four main principles that we followed when porting this application on the TaihuLight.

- BFS is a data intensive application, so we should maximize the utilization of both memory and network bandwidth by batching the data.
- Inevitable random data access should be restricted in a relatively faster device to minimize memory access overhead.
- To reduce the waiting time caused by the intrinsic data dependency of BFS, data should be transmitted or processed as soon as it is ready.
- Atomic operations should be avoided as much as possible due to their large overhead.

4.2. Pipelined Module Mapping

TaihuLight has two types of heterogeneous processor cores, i.e., CPE and MPE, which have different functionalities and computing capacities. To implement a highly efficient asynchronous BFS framework on this heterogeneous architecture, we propose a pipelined module mapping technique. The basic idea is to map different modules of the BFS algorithm into different CPE clusters to leverage the high bandwidth of CPE clusters and use a dedicated MPE to deal with network communication and scheduling work exclusively.

As shown in Figure 1, we have split the BFS algorithm into several processing modules, which should run in CPE clusters concurrently. In a node, we name four MPEs and four CPE clusters as M0/M1/M2/M3 and C0/C1/C2/C3 respectively. MPEs are dedicated to inter-node communication and management work, while CPE clusters are in charge of the main work of each module.

We use two nodes (node0 and node1) to illustrate our mapping strategy for the Top-Down and Bottom-Up traversals. In the Top-Down traversal of Figure 4(a), we map *Forward Generator* module into C0 of node1 and *Forward Handler* module into C3 of node0. When C0 in node1 generates enough messages, C0 notifies M0 to send the messages to a destination node (e.g., node0). All the MPI work is done on the MPE and no memory copy operations are needed due to the unified memory view of the MPEs and CPEs. When receiving a message, M1 in node0 schedules an idle CPE cluster (e.g., C3) to process the forward messages.

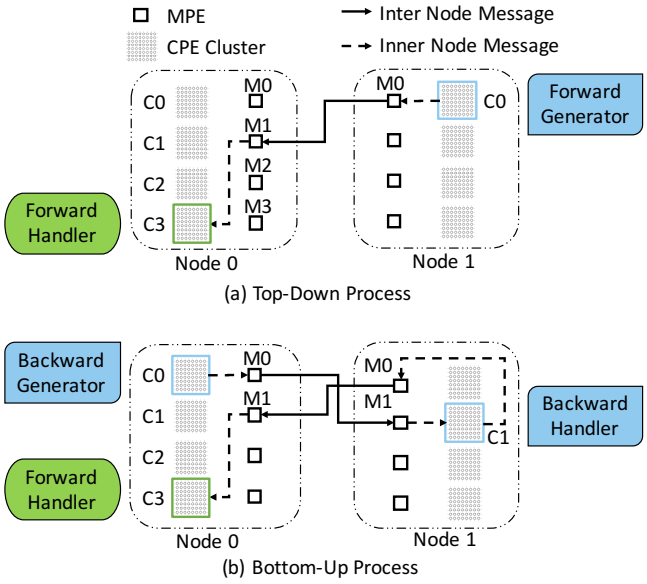

Figure 4: Pipelined module mapping among heterogeneous units. In a node, four MPEs are named as M0/M1/M2/M3, and CPE clusters are named as C0/C1/C2/C3. We use M0 and M1 to send and receive messages, respectively. C0/C1/C2/C3 are used to process modules whenever one is available. (a) and (b) show the Top-Down and Bottom-Up traversals respectively using the pipelined module mapping technique.

Since MPEs and CPEs have their own clocks and PCs, the notifications between the MPEs and CPEs use a busy-wait polling mechanism by checking flags set in memory. Inside a CPE cluster, we use a more efficient mechanism, i.e., register communication, to pass the flag. For example, when an MPE notifies a CPE cluster, the MPE sets a flag (corresponding module function) in memory of a representative CPE in the cluster. Then the representative CPE gets the notification in memory and broadcasts the flag to all other CPEs in the cluster.

The Bottom-Up traversal is similar to the Top-Down traversal. In node0, the *Backward Generator* and *Forward Handler* modules are mapped into C0 and C3, respectively. The data transmission path in the Bottom-Up traversal is longer than the Top-Down traversal due to more modules, as shown in Figure 4(b).

To avoid data contention and starvation, in our design, no more than one CPE cluster executes the same module in one node at any time. We will discuss this further it in Section 4.4, where all the modules are described.

4.3. Contention-Free Data Shuffling

In this subsection, we show how to deal with the data shuffling of modules within a CPE cluster. In Section 2, we have described two types of module, namely the reaction module and the dispose module. The latter one can be applied in a CPE cluster by partitioning the input data for different CPEs, while the former one is much more complex. As discussed in Section 2.1, the reaction module produces data to be sent to a dedicated destination in a random manner. Exacerbating the issue, the data can be generated at any time under an asynchronous framework. We will show how to employ a contention-free data shuffling technique to generate the data and efficiently write it to the corresponding sending buffer in batches.

A straightforward implementation for CPEs collaboration is to partition the input data for different CPEs. Then, the data randomly generated may be written to the same sending buffer, which requires atomic operations to guarantee coherence. However, atomic operations in the main memory are inefficient and have bad performance in the current TaihuLight system.

To avoid atomic operations, we leverage the on-chip register communication buses of the TaihuLight. We designed a communication mechanism in which different CPEs can coordinate their actions and share their local memory, rather than working separately.

Another problem arises when applying the above mechanism. If each CPE is responsible for a set of destinations, there will be random messages between any pair of CPEs. However, register communication uses a mesh network and explicit synchronous interfaces, which means that deadlock is inevitable.

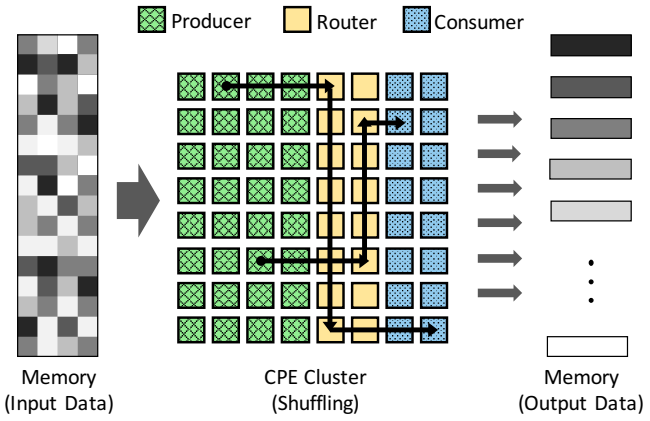

Figure 5: Data shuffling within a CPE cluster. Data is read in batches from memory, then processed and shuffled in a CPE cluster, written back into memory and sorted without contention. This technique is suitable for all reaction modules.

We propose a technique by carefully assigning three different roles to CPEs: producer, consumer, and router. Producers are responsible for reading data from memory and sending messages to routers using the register bus. Although the specific work for the producer of different modules varies, they all generate batched data in the same way. Routers are responsible for shuffling, e.g., checking the content of the packet, deciding on the right destination, and transmitting them to consumers. The rule is that all data that should eventually be written to the global memory space

goes to the same consumer. Consumers only buffer the data received and write it back to memory in batches.

We use *Forward Generator* module as an example, whose pseudo-code is function FORWARD_GENERATOR in Algorithm 2. the $Curr$, i.e., current frontier, is parallelized among producers. Each producer processes the assigned frontier vertices, i.e., u, by checking each neighbor of the vertices, i.e., v, then generates the data pairs, i.e., (u, v). Each data pair is a message needs to shuffle, the destination producer is determined by the value of v.

Within a module, each producer processes the assigned data and generates messages to be sent to specific nodes, where each node corresponds to a consumer. To satisfy the mesh network of the register communication, routers are used to deliver the messages to the consumer. Figure 5 illustrates a working assignment. The blocks are CPEs, and the small arrows aside represent the forward direction. In our solution, a column always passes messages in one direction. The first four columns of producers pass from left to right, dealing with the heaviest part of module. There are two columns of routers for upward and downward pass, which is necessary for deadlock-free configuration. The last two columns only consume data and guarantee the output throughput. A deadlock situation cannot arise if there is no circular wait in the system.

This design has several advantages. First, global memory reads and writes are issued by producers and consumers. All the operations can be done in a batched DMA fashion, which is orders of magnitude faster than fragmented random access. Second, the strong demand for local memory size in one single CPE is distributed to 64 CPEs in the same cluster. Taking consumers as an example, if we have 16 consumers, $16 \times 64KB$ can be used as a buffer for total memory space. Using 256 Bytes as the batch size, we can handle up to 1024 destinations in practice, and Section 4.4 explains how to extend it to 40,000. Based on our communication protocol, consumers can write to different global memory locations and therefore incur no overhead in atomic operations. In general chip design and manufacturing, it is always difficult and expensive to insert more SRAM space into weak cores. The communication model proposed simply tackles this problem simply.

Finally, the number of producers, routers and consumers depends on specific architecture details. Specifically, DMA read bandwidth, DMA write bandwidth, CPE processing rate, and register bus bandwidth together determine the final count. Combining all the discussions, in a micro-benchmark, we achieve 10 GB/s register to register bandwidth out of a theoretical 14.5 GB/s (half of peak bandwidth in Figure 3 for both read and write), which has a significant impact on overall performance. The evaluation section provides more detailed results.

We will discuss the utilization of MPEs and CPEs after introducing the other two modules in Section 4.4.

4.4. Group-Based Message Batching

When the node number becomes extremely large, about 40,000 nodes in the TaihuLight, the peer-to-peer random messages in an asynchronous BFS framework increase dramatically. Apart from the levels with extremely large frontier, i.e., the middle one or two levels of a ceratin BFS run, the message size in other BFS levels is quite small, usually less than 1 KB in most cases, resulting in inefficient bandwidth usage.

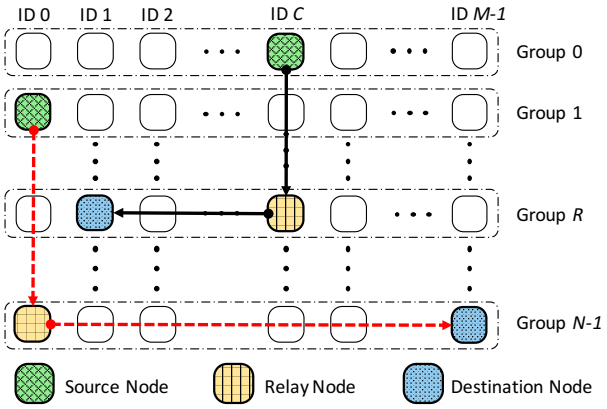

Figure 6: Group-based message batching. The machine's nodes are arranged as an N $\times$ M matrix. Each message uses a relay node that is in the same row as the destination node and the same column as the source node.

To solve this problem, we arrange the nodes into an N $\times$ M matrix, that is, N groups, where each group has M nodes. We split peer-to-peer messaging into two stages. In stage one, the message is first sent to the relay node that is located in the same row as destination node and the same column as the source node. In stage two, the relay node sends the message to the destination node. The process is illustrated in Figure 6.

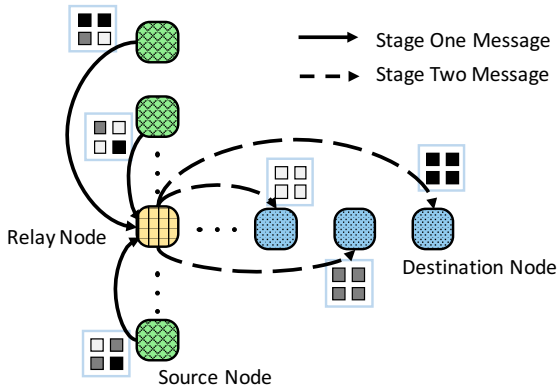

Figure 7: A batching example of group-based message batching technique. The relay node uses a *Relay Module to shuffle the data from stage one, generating data to be sent in stage two.*

The batching of messages happens in two ways, shown in Figure 7. In stage one, the messages to the same group are batched in the source node and sent together, whereas the receiving relay node shuffles the batched messages and

stores them to the related destination buffer for stage two usage. The relay node gathers all the messages in the same column and sends them together in stage two. Since every node can be a source, relay, or destination node, there is no obvious bottleneck.

Supposing each pair of nodes sends and receives a message small enough, every node will send (N × M) messages, equal to the amount of nodes. Applying our technique, the message number is only (N + M - 1) (N messages to the same column plus M messages to the same row, minus the node itself), which is a dramatic reduction.

Further, suppose that each connection reserves 100 KB memory and the total node number is 40,000, the number of connections for every MPE is reduced from (N × M) to (N + M - 1), means the MPI library memory overhead is reduced from (40000 × 100 KB =) 4 GB to ((200 + 200 - 1) * 100 KB =) 40 MB, approximately.

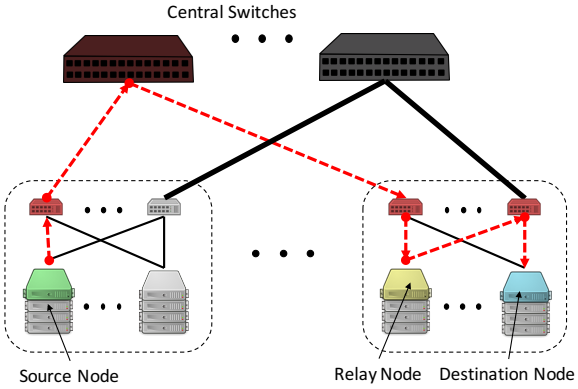

Figure 8: Logical and physical group mapping. The logical group is mapped to a super node in the fat-tree topology.

To cancel out the bandwidth overhead further, we map each communication group into the same super node. Figure 8 shows the route a message travels. A test is designed to calculate the additional relay operation overhead. We compare the speed of sending only relatively big messages only to the relay node and having the messages sent to the destination node, through the relay node, to determine whether this two-step messaging system would cause additional overhead. The results show that no bandwidth difference between the two settings exists, as both achieve an average 1.2 GB/s per node. This may be because the central network is capped at one fourth of the maximum bisection bandwidth, resulting in the bandwidth within the super node being four times higher than the central network, and the relay operation being hidden by the higher super node network.

Each relay node includes a shuffling operation with produced data sent to other nodes; therefore, these are also reaction modules. We add the module in both forward and backward routes, namely *Forward Relay* and *Backward Relay*. Figure 1 shows the complete modules of the Top-Down and Bottom-Up traversals.

After the two new modules are added, the pipeline module mapping becomes that shown in Figure 9. We use

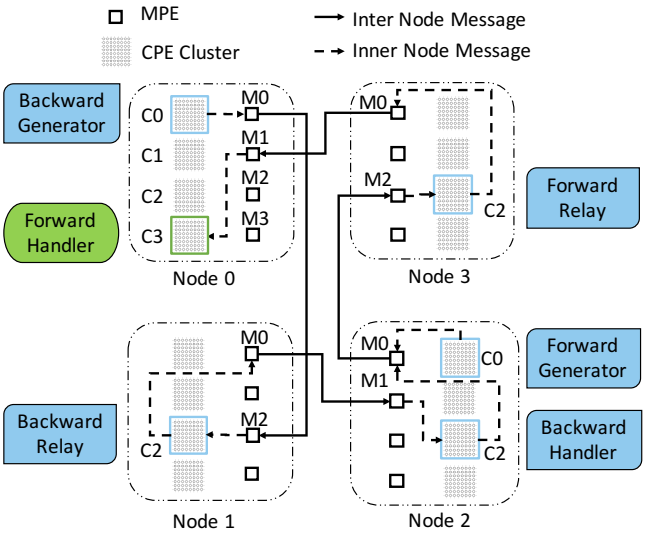

Figure 9: Complete version of pipelined module mapping among heterogeneous units. *Forward Relay* and *Backward Relay* modules are added into both forward and backward path. The Top-Down traversal and Bottom-Up traversal share some modules only in functionality.

a first-come-first-serve module scheduling policy. There are five modules in the Bottom-Up traversal in total, however, which is larger than the amount of physical CPE clusters in a node. A simple but efficient method to solve the probable deadlock when all the CPE clusters are busy is to process the module in the MPE instead. According to the profile results, this rarely occurs because the local processing speed in CPEs is faster, on average, than the network communication speed.

5. Implementation

We use the MPI plus a multi-thread programming model, in which MPI runs only on MPEs, and CPEs using a pthread-like interface. The following are some implementation details.

Degree aware prefetch To improve load balance on hub vertices, which have much large degree of most other vertices, we prefetch the frontiers of all the hub vertices on every node to avoid possible network communication at runtime. We choose a fixed number of hub vertices per node (2^{12} for Top-Down, 2^{14} for Bottom-Up), and a bitmap is used for compressing the frontiers. This method is based on prior work on combined 1D/2D partitioning [4], [10].

Quick processing for small messages If the input of a module is small enough, the work is done in the MPE directly instead of sending it to a CPE cluster. We set the threshold to 1 KB, which is calculated based on the notification overhead and the memory access ability difference between the MPEs and the CPE clusters.

Reduce global communication The program gathers frontier data from every hub vertex at every level. The cost

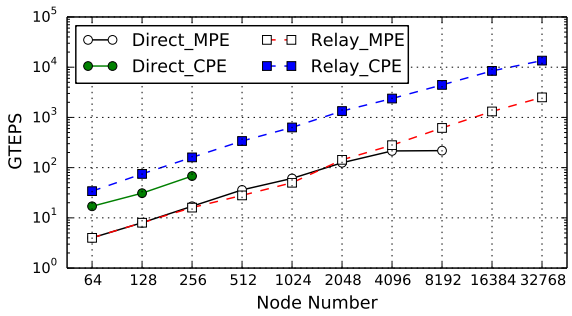

Figure 10: Performance comparison of techniques. The four lines symbolize different implementations with different techniques. The tags *Relay* and *Direct* indicate whether a group-based message batching technique in Section 4.4 is used. The line with the *CPE* tag shows the result with the contention-free data shuffling technique in Section 4.3, while the *MPE* tag indicates the modules are processed in the MPE directly. The vertex number of each node on average is 16 million.

of this operation increases linearly to node numbers. This operation does not scale well, and we tried to reduce the cost of this operation as much as possible. If the hub vertices frontier is empty, which happens for a number of later levels, a special flag is gathered instead of a bitmap.

Customized global communication To make use of the customized 2-level fat-tree network, a group based implementation of *MPI_Allgather* is developed, which gathers the data within groups, then across groups. In the case of gathering 2 KB sized messages from 32,768 nodes, this method reduces the cost of *MPI_Allgather* from 0.14s to 0.04s, which is about 68.8% faster than to the default implementation in SWMPI 2.2 (Mvapich 2.2rc1 based).

In addition to the above implementation, we also balance the graph partitioning and optimize the BFS verification algorithm to scale the entire benchmark to 10.6 million cores.

6. Evaluation

The Table 1 shows the platform specification. The C and MPI compiler used are Sunway REACH (Open64 based) and SWMPI 2.2 (Mvapich 2.2 based), respectively. Our framework conforms to the Graph500 benchmark specifications using the Kronecker graph raw data generator, and the suggested graph parameter, that is, the edge factor, is fixed to 16.

6.1. Impact of Techniques

We have designed an experiment to evaluate the performance of some of the techniques proposed in Section 4. The contention-free data shuffling technique is compared by modules processed in the MPE, whose performance is better than lock-based CPE cluster implementation. Relay version of the implementation using the group-based message batching technique is compared to the direct messaging version. Figure 10 presents the experimental results.

Comparing *Direct_MPE* with *Direct_CPE*, or *Relay_MPE* with *Relay_CPE*, we can conclude that properly used CPE clusters can improve performance by a factor of 10. The shuffling within the CPE cluster technique has a better performance for up to 256 nodes, but it crashes when the scale increases because of the limitation of SPM size on the CPEs. Splitting network communications into inner and inter groups makes this technique functional at larger scales.

The scalability of *Direct_MPE* is initially linear, but is capped at 4,096 nodes. This is because too many small messages limit the bandwidth when every message is directly sent to the destination nodes. At a scale of 16,384 nodes, Direct MPE crashes from memory exhaust caused by too many MPI connections.

The *Relay_CPE* is our final version, combining all the techniques, which scales with good performance to run on the whole machine.

6.2. Scaling

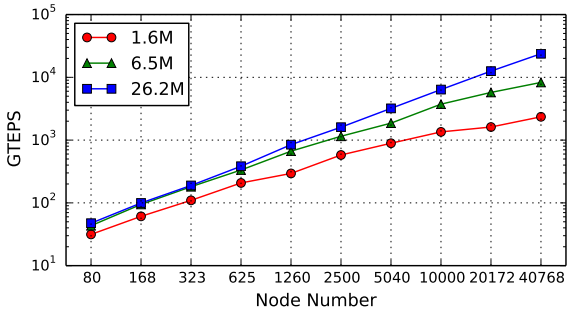

Figure 11: Weak scaling of BFS by fixing the vertices number per node and increase the node number. Different lines represent different average vertices numbers of each node.

This part reports the weak scaling result. Three different data sizes of data are tested, and the per node vertices numbers are set as *1.6M*, *6.5M* and *26.2M*, which leads to the vertices numbers of whole 40,768 nodes being 2^{36}, 2^{38} and 2^{40}, respectively.

From Figure 11, we find that the result has almost linear weak scaling speedup with the CPU number increasing from 80 to 40,768, owing to the techniques discussed in Section 4.

Although the lines share a similar starting point, the performance diverges as the node number increases. At the scale of 40,768, the result of *26.2M* is nearly four times that of *6.5M*, with the same gap between *6.5M* and *1.4M*. We have proposed a group-based batching technique to batch the massive amounts of small messages, but it is still not large enough to exhaust the network bandwidth. When data size is small, such as *1.6M* and *6.5M*, as shown in Figure 11, the high latency is the main reason for inefficiency.

Authors	Year	Scale	GTEPS	Num Processors	Architecture	Homo. or Hetero.
Ueno [11]	2013	35	317	1,366 (16.4K Cores) + 4096	Xeon X5670 + Fermi M2050	Hetero.
Beamer [3]	2013	35	240	7,187 (115.0K Cores)	Cray XK6	Homo.
Hiragushi [12]	2013	31	117	1,024	Tesla M2090	Hetero.
Checconi [4]	2014	40	15,363	65,536 (1.05M Cores)	Blue Gene/Q	Homo.
Buluc [5]	2015	36	865.3	4,817 (115.6K Cores)	Cray XC30	Homo.
(K Computer) [2]	2015	40	38,621.4	82,944 (663.5K Cores)	SPARC64 VIIIfx	Homo.
Bisson [13]	2016	33	830	4,096	Kepler K20X	Hetero.
Present Work	2016	40	**23,755.7**	**40,768 (10.6M Cores)**	SW26010	Hetero.

TABLE 2: Results of recent works implementing BFS on distributed systems. The result of K Computer from the Graph500 website is also included [2], although no paper has been published yet.

6.3. Breakdown

We separate the BFS algorithm into several components to analyze time breakdown, including *Top-Down Traversal*(27%), *Bottom-Up Regular Traversal*(49%) and *Bottom-Up Hub Traversal*(24%), where *Bottom-Up Traversal* is divided into two parts. The asynchronous framework incurs a heavy percentage for both Top-Down and Bottom-Up. Additionally, the local hub vertices process can not be ignored.

6.4. Comparison with Other Work

Table 2 lists recently published results of distributed BFS work in recent years including both homogeneous and heterogeneous architecture. Our work is among the largest scale and achieves the best result for heterogeneous architecture.

7. Related Work

In recent years, analyzing big data has gained tremendous attention. BFS, one of the most representative algorithms among the applications, has attracted many researchers.

Load balance is one of the most challenging problems researchers face when implementing BFS on a heterogeneous architecture. Scarpazza et al. implement BFS in Cell/BE by splitting data into chunks to fit the data into local storage with synergistic processing elements [14]. On GPU, one simple approach is mapping vertices in the frontier to discrete threads [15], which suffers from a serious load balance problem. Hong et al. propose a method to somewhat balance the process by assigning a warp with a set of vertices [16], while Merrill et al. carefully optimize neighbor-gathering and status-lookup stage within a Cooperative Thread Array (CTA) [17]. Other accomplishments, such as [18], [19], use thread, warp, CTA or the whole processor to deal with vertices of different degrees. We are inspired from these pioneers and present a solution suitable to Sunway's unique heterogeneous architecture.

When the size of the data increases, the memory of a single CPU is insufficient. Previously, efforts have been put into using multi-GPUs [17], [19], a collaborating CPU-GPU design [20], and external storage [21], [22] to extend the ability of their programs.

Some attempt to maximize the performance of the BFS algorithm on a single machine. Optimizing methods include using bitmaps or a hierarchy structure for frontiers to restrict random access in a faster cache [23], [24], lock-free array updates [24], and serializing the random access among NUMA [23].

Beamer et al. use a direction optimization method that enables Bottom-Up traversal with traditional Top-Down traversal, which largely decreases the memory access number and also improves load balance [7]. This method opens an interesting path to optimizing BFS. Yasui et al. give an example of how to refine graph data structure to speed up Bottom-Up traversal [25].

The distributed BFS algorithm can be divided into 1D and 2D partitioning in terms of data layout [26]. Buluc et al. discuss the pros and cons of 1D and 2D partitioning in their paper [6]. Based on the specific 5D torus network of BlueGene/Q, Checconi et al. overlap the stages of 2D partitioning and achieve the best performance at that time [27]. The most similar work to ours is the 1D partitioning with direction optimization given in [4]. However, the difference in CPU architecture and network cause our designs and implementations to diverge. Other than, attempts have been made to make use of heterogeneous devices in a distributed environment [11]–[13]. However, they failed to achieve adequate results for the reason direction optimization method is not included.

Pearce et al. have some discussion of how to reduce the number of communication channels per process and aggregate messages in a torus network in BlueGene [22], while our group-based message batching technique focuses on a large over-subscribed fat-tree network.

Message compression is also an important optimization method [4], [27], [28], which is orthogonal to our work. It may be integrated with our work in future.

8. Discussion

The techniques proposed in this paper include both micro-architecture dependent and independent optimizations. Among the three techniques we proposed in the paper, the pipelined module mapping and the group-based message batching can be applied to other large-scale heterogeneous architectures in a similar way, while the contention-free data shuffling technique is quite micro-architecture dependent.

The key operations of the distributed BFS can be viewed as shuffling dynamically generated data, which is also the major operations of many other graph algorithms, such as Single Source Shortest Path (SSSP), Weakly Connected Component (WCC), PageRank, and K-core decomposition. All the three key techniques we used are readily applicable to other irregular graph algorithms.

9. Conclusion

We share our experience of designing and implementing the BFS algorithm on Sunway TaihuLight, which is a large scale parallel computer with heterogeneous architecture. We ultimately achieved running the program on 40,768 nodes, nearly the whole machine, with a speed of 23,755.7 GTEPS, which is the best among heterogeneous machines and second overall in the Graph500s June 2016 list [2].

Acknowledgment

We would like to thank Tianjian Guo and the colleagues in PACMAN group. We also thank the anonymous reviewers for their insightful comments and guideline. This work is partially supported by NSFC Distinguished Young Scholars Grant No. 61525202 and National Key Research and Development Program of China 2016YFB0200100. The corresponding author is Wenguang Chen (Email: cwg@tsinghua.edu.cn).

References

[1] Top500, "http://www.top500.org/."

[2] Graph500, "http://www.graph500.org/."

[3] S. Beamer, A. Buluc, K. Asanovic, and D. Patterson, "Distributed memory breadth-first search revisited: Enabling bottom-up search," *Proceedings of IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW '13*, pp. 1618–1627, 2013.

[4] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS '14*, no. December 2010, pp. 425–434, 2014.

[5] A. Buluc, S. Beamer, K. Madduri, K. Asanović, and D. Patterson, "Distributed-Memory Breadth-First Search on Massive Graphs," in *Parallel Graph Algorithms*, 2015.

[6] A. Buluc and K. Madduri, "Parallel breadth-first search on distributed memory systems," *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11*, pp. 1–12, 2011.

[7] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, vol. 21, no. 3-4, 2012, pp. 137–148.

[8] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang, "The Sunway TaihuLight supercomputer: system and applications," *Science China Information Sciences*, vol. 072001, 2016.

[9] J. Dongarra, "Report on the Sunway TaihuLight System," *Tech Report UT-EECS-16-742*, p. 2016, 2016.

[10] R. Pearce, M. Gokhale, and N. M. Amato, "Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates," *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14*, pp. 549–559, 2014.

[11] K. Ueno and T. Suzumura, "Parallel distributed breadth first search on GPU," in *Proceedings of 20th Annual International Conference on High Performance Computing, HiPC'13*, 2013, pp. 314–323.

[12] T. Hiragushi and D. Takahashi, "Efficient hybrid breadth-first search on GPUs," in *ICA3PP' 13*, vol. 8286 LNCS, no. PART 2, 2013, pp. 40–50.

[13] M. Bisson, M. Bernaschi, and E. Mastrostefano, "Parallel Distributed Breadth First Search on the Kepler Architecture," *Proceedings of IEEE Transactions on Parallel and Distributed Systems, TPDS '16*, vol. 27, no. 7, pp. 2091–2102, 2016.

[14] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/BE processor," *Proceedings of IEEE Transactions on Parallel and Distributed Systems, TPDS '08*, vol. 19, no. 10, pp. 1381–1395, 2008.

[15] L. L. L. Luo, M. Wong, and W.-m. H. W.-m. Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of Design Automation Conference, DAC '10*, 2010, pp. 52–55.

[16] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP '11*, vol. 46, no. 8, 2011, p. 267.

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

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

[19] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16*. New York, New York, USA: ACM Press, 2016, pp. 1–12.

[20] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," *Proceedings of Parallel Architectures and Compilation Techniques, PACT '11*, pp. 78–88, 2011.

[21] R. Korf and P. Schultze, "Large-scale parallel breadth-first search," *AAAI '05*, pp. 1380–1385, 2005.

[22] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," *Proceedings of IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS '13*, pp. 825–836, 2013.

[23] V. Agarwal, D. Pasetto, and D. A. Bader, "Scalable Graph Exploration on Multicore Processors," *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10*, no. November, 2010.

[24] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS '12*, 2012, pp. 378–389.

[25] Y. Yasui and K. Fujisawa, "Fast and scalable NUMA-based thread parallel breadth-first search," in *Proceedings of 2015 International Conference on High Performance Computing & Simulation, HPCS '15*. IEEE, jul 2015, pp. 377–385.

[26] A. Yoo and K. Henderson, "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene / L," *Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05*, p. 25, 2005.

[27] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC '12*, 2012.

[28] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration," *Proceedings of 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors ASAP 12*, pp. 8–15, 2012.